Светлин Наков
и колектив

ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО

с Java

SOFTWARE UNIVERSITY

SOFTWARE UNIVERSITY FOUNDATION

Кратко съдържание

Въведение в програмирането с Java

Светлин Наков и колектив

Борис Вълков
Веселин Колев
Владимир Цанев
Данаил Алексиев
Лъчезар Божков
Лъчезар Цеков
Марин Георгиев
Марио Пешев
Мариян Ненчев
Михаил Стойнов
Николай Василев
Николай Недялков
Петър Велев
Радослав Иванов
Румяна Топалска
Стефан Стаев
Светлин Наков
Теодор Стоев
Христо Тодоров
Цвятко Конов

София, 2008

Въведение в програмирането с Java

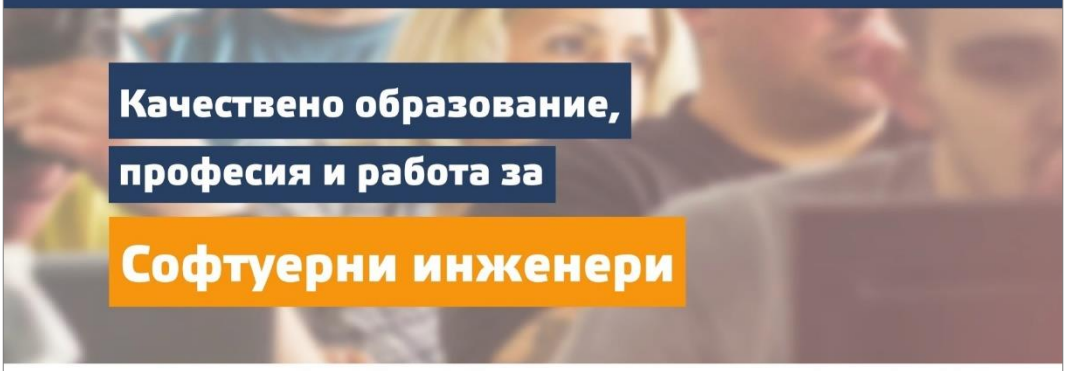
Първо издание. Последна редакция: април 2017 г.

Официален уеб сайт:

http://www.introprogramming.info

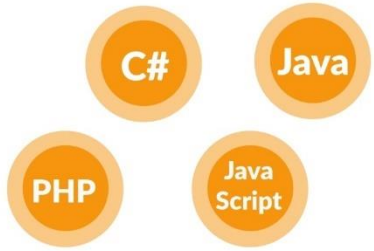
ISBN 978-954-400-055-4

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

Съдържание

Предговор

Ако искате да се захванете сериозно с програмиране, попаднали сте на **правилната книга**. Наистина! Това е книгата, с която можете да направите първите си стъпки в програмирането. Тя ще ви даде **солидни основи** от знания, с които да поемете по дългия път на изучаване на съвременните езици за програмиране, платформи и технологии за разработка на софтуер. Това е книга за програмиране, която учи на **фундаменталните концепции за разработка на компютърни програми**, които не са се променили съществено през последните 15 години.

Не се притеснявайте да прочетете тази книга, дори **Java** да не е езикът, с който искате да се занимавате. С който и друг език да продължите по-нататък, знанията и уменията, които ще ви дадем, ще ви останат трайно, защото тази книга ще ви научи **да мислите като програмисти**. Ще ви покажем как да пишете програми, с които да решавате практически задачи по програмиране, ще ви научим **да измисляте и реализирате алгоритми** и да ползвате различни структури от данни.

Колкото и да ви се струва невероятно, базовите принципи на писане на компютърни програми не са се променили съществено през последните 15 години. Езиците за програмиране се променят, технологиите се променят, средствата за разработка се развиват, но **принципите на програмирането си остават едни и същи**. Когато човек се научи да мисли алгоритмично, когато се научи инстинктивно да разделя проблемите на последователност от стъпки и да ги решава, когато се научи да подбира подходящи структури от данни и да пише качествен програмен код, тогава той става програмист. Когато придобиете тези умения, лесно **можете да научите нови езици** и различни технологии, като уеб програмиране, бази от данни, мобилни технологии, HTML, XML, SQL и още стотици други.

Тази книга е именно за това да ви научи **да мислите като програмисти**, а езикът Java е само един инструмент, който може да се замени с всеки друг съвременен език. **Това е книга за програмиране, а не книга за Java!** Тя ще ви даде концепции за разработка на компютърни програми, а не просто някакви знания за един конкретен програмен език.

За кого е предназначена тази книга?

Тази книга е **за начинаещи**. Тя е предназначена за всички, които не са се занимавали до момента сериозно с програмиране и имат желание да започнат. Тази книга **започва от нулата** и ви запознава стъпка по стъпка с основните на програмирането. Тя няма да ви научи на всичко, което ви трябва, за да станете софтуерен инженер и да работите в софтуерна фирма,

но ще ви даде **основи, върху които да градите** технологични знания и умения, а с тях вече ще можете да превърнете програмирането в професия.

Ако никога не сте писали компютърни програми, не се притеснявайте. Винаги има първи път. В тази книга **ще ви научим на програмиране от нулата**. Не очакваме да знаете и можете нещо предварително. Достатъчно е да имате компютърна грамотност и желание да се занимавате с програмиране. Останалото ще го прочетете от тази книга.

Ако вече можете да пишете прости програмки или сте учили програмиране в училище или в университета или сте писали програмен код с приятели, не си мислете, че знаете всичко! Прочетете тази книга и ще се убедите **колко много неща сте пропуснали**. Книгата е за начинаещи, но ви дава концепции, които дори някои програмисти с опит не владеят. По софтуерните фирми са се навъдили възмутително много самодейци, които, въпреки, че програмират на заплата от години, не владеят основите на програмирането и не знаят какво е **хеш-таблица**, как работи **полиморфизмът** и как се работи с **битови операции**. Не бъдете като тях! Научете първо основите на програмирането, а след това технологиите. Иначе рискувате **да останете осакатени като програмисти** за много дълго време (а може би и за цял живот).

Ако път имате опит с програмирането, **за да прецените дали тази книга е за вас**, я разгледайте подробно и вижте дали са ви познати всички теми, които сме разгледали. Обърнете особено внимание на главите "Структури от данни (линейни, дървовидни, хеш-таблици)", "Принципи на обектно-ориентираното програмиране", "Как да решаваме задачи по програмиране?" и "Качествен програмен код". Много е вероятно дори ако имате няколко години опит, да не владеете добре работата със **структури от данни**, концепциите на **обектно-ориентираното програмиране** (включително UML и design patterns) и да не познавате добрите практики за писане на **качествен програмен код**. Това са много важни теми, които не се срещат във всяка книга за програмиране!

Не са необходими начални познания

В тази книга не очакваме от читателите да имат предварителни знания по програмиране. Не е необходимо да сте учили информационни технологии или компютърни науки, за да четете и разбирате учебния материал. **Книгата започва от нулата** и постепенно ви въвлича в програмирането. Всички технически понятия, които ще срещнете, са обяснени преди това и не е нужно да ги знаете от друго място. Ако не знаете какво е компилатор, дебъгер, среда за разработка, променлива, масив, цикъл, конзола, символен низ, структура от данни, клас или обект, не се притеснявайте. Ще научите всички тези понятия и много други и ще се научите да ги ползвате непрестанно в ежедневната си работа. Просто четете книгата последователно и **непременно решете задачите след всяка тема**.

Не се очаква да имате познания по компютърни науки и информационни технологии, но ако все пак имате такива, при всички

положения ще са ви от полза. Ако учите университетска специалност, свързана с компютърните технологии или в училище учите информационни технологии, това **само ще ви помогне, но не е задължително**. Ако учите туризъм или право, също можете да станете добър програмист, стига да имате желание.

Би било полезно да имате **начална компютърна грамотност**, тъй като няма да обясняваме какво е файл, какво е твърд диск, какво е мрежова карта, как се движи мишката и как се пише на клавиатурата. Очакваме да знаете как да си служите с компютъра и как да ползвате Интернет.

Препоръчва се читателите да имат някакви **знания по английски език**, поне начални. Всичката документация, която ще ползвате ежедневно, и почти всички сайтове за програмиране, които ще четете постоянно, са на английски език. В професията на програмиста **английският е просто задължителен**. Колкото по-рано го научите, толкова по-добре.

Не си правете илюзии, че можете да станете програмисти, без да научите поне малко английски език! Това е просто наивно очакване. Ако не знаете английски, започнете да четете технически текстове и си вадете непознатите думи и ги заучавайте. Ще видите, че техническият английски се учи лесно и не отнема много време.

Какво обхваща тази книга?

Настоящата книга обхваща **основите на програмирането**. Тя ще ви научи как да дефинирате и използвате променливи, как да работите с примитивни структури от данни (като например числа), как да организирате логически конструкции, условни конструкции и цикли, как да печатате на конзолата, как да ползвате масиви, как да работите с бройни системи, как да дефинирате и използвате методи и да създавате и използвате обекти.

Наред с началните познания по програмиране книгата ще ви помогне да възприемете и малко **по-сложни концепции** като обработка на символни низове, работа с изключения, използване на сложни структури от данни (като списъци, дървета и хеш-таблици), работа с текстови файлове и дефиниране на собствени класове. Ще бъдат застъпени в дълбочина концепциите на **обектно-ориентираното програмиране** като утвърден подход при съвременната разработка на софтуер. Накрая ще се сблъскате с практиките за **писане на висококачествени програми** и с решаването на реални проблеми от програмирането. Книгата излага **цялостна методология за решаване на задачи** по програмиране и въобще на алгоритмични проблеми и показва как се прилага тя на практика с няколко примерни теми от изпити по програмиране. Това е нещо, което няма да срещнете в никоя друга книга за програмиране.

На какво няма да ви научи тази книга?

Тази книга няма да ви даде професията "софтуерен инженер"! Тази книга **няма да ви научи да ползвате цялата Java платформа**, да работите с бази от данни, да правите динамични уеб сайтове и мобилни приложения и да боравите с прозоречен графичен потребителски интерфейс. Няма да се научите да пишете сериозни софтуерни приложения като Skype, Firefox или Angry Birds за телефон. За това са нужни **много, много години работа**.

От книгата **няма да се научите софтуерно инженерство** и работа в екип и няма да можете да се подготвите за работа по реални проекти в софтуерна фирма. За да се научите на всичко това ще ви трябват **още няколко книги и допълнителни обучения**, но не съжалявайте. Правите правилен избор като започвате от основите на програмирането вместо директно от уеб приложения. Това ви дава шанс да станете добър програмист, който разбира технологиите в дълбочина. След като усвоите **основите на програмирането**, ще ви е много по-лесно да четете за бази данни, уеб и мобилни приложения и ще разбирате това, което четете, много по-лесно, отколкото, ако се захванете директно със SQL, JavaScript, PHP или ASP.NET.

Някои ваши колеги започват да програмират директно от уеб приложения и бази от данни, без да знаят какво е масив, какво е списък и какво е хеш-таблица. Не им завиждайте! Те са тръгнали по **трудния път, отзад напред**. И вие ще научите тези неща, но преди да се захванете с тях, се научете да програмирате. Това е много по-важно. Да научите една или друга технология е много по-лесно, **след като имате основата**, след като можете да мислите алгоритмично и знаете как да подхождате към проблемите на програмирането.

Да започнете с програмирането от уеб приложения и бази данни е също толкова неправилно, колкото и да започнете да учите чужд език от някой класически роман вместо от буквар или учебник за начинаещи. Не е невъзможно, но като ви липсват основите, е много по-трудно. Възможно е след това с години да останете без важни фундаментални знания и да ставате за смях на колегите си.

Как е представена информацията?

Въпреки големия брой автори, съавтори и редактори, стилът на текста в книгата е изключително достъпен. Съдържанието е представено в **добре структуриран вид**, разделено с множество заглавия и подзаглавия, което позволява лесното му възприемане, както и бързото търсене на информация в текста.

Настоящата книга е **написана от програмисти за програмисти**. Авторите са действащи софтуерни разработчици, хора с реален опит както в разработването на софтуер, така и в обучението по програмиране. Благодарение на това качеството на изложението е на много високо ниво.

Всички автори ясно съзнават, че примерният сорс код е едно от най-важните неща в една книга за програмиране. Именно поради тази причина текстът е съпроводен с **много, много примери, илюстрации и картинки**.

Няма как, когато **всяка глава е писана от различен автор**, да няма разминаване между стиловете на изказ и между качеството на отделните глави. Някои автори вложиха много старание (месеци наред) и много усилия, за да станат **перфектни** техните глави. Други не вложиха достатъчно усилия и затова някои глави не са така хубави и изчерпателни като другите. Не на последно място опитът на авторите е различен: някои програмират професионално от 1-2 години, докато други – от 15 години насам. Няма как това да не се отрази на качеството, но ви уверяваме, че всяка глава е минала редакция и отговаря поне минимално на високите изисквания на водещия автор на книгата – Светлин Наков.

Какво е Java?

Вече обяснихме, че **тази книга не е за Java, а за програмирането** като концепция и основни принципи. Ние използваме езика и платформата Java само като средство за писане на програмен код и не наблягаме върху спецификите на езика. Настоящата книга може да бъде намерена и във варианти за други езици като C# и C++, но разликите не са съществени.

Все пак, нека разкажем с няколко думи **какво е Java**.

Java е съвременен език за програмиране и платформа за разработка и изпълнение на софтуерни приложения.

Някои си мислят, че Java е само език за програмиране, други го бъркат с JavaScript и си мислят, че служи за раздвижване на статични уеб сайтове, а трети се чудят дали Java е сорт кафе или остров в Индонезия. За да разсеем съмненията, ще разкажем с няколко думи за **езика и платформата Java**, а в следващата глава ще научите за тях много повече.

Езикът Java

Java е съвременен обектно-ориентиран език за програмиране с общо предназначение. На Java се разработва изключително разнообразен софтуер: офис приложения, уеб приложения, настолни приложения, приложения за мобилни телефони, игри и много други.

Java е език от високо ниво, който прилича на C# и C++ и донякъде на езици като Delphi, VB.NET и C. Java програмите са обектно-ориентирани. Те представляват съвкупност от дефиниции на класове, които съдържат в себе си методи, а в методите е разположена програмната логика. Повече детайли за това какво е клас, какво е метод и какво представляват Java програмите ще научите в следващата глава.

В днешно време **Java е един от най-популярните езици за програмиране**. На него пишат милиони разработчици по цял свят. Най-

големите световни софтуерни корпорации като **IBM**, **Oracle**, **Google** и **SAP** базират много от своите решения на Java платформата и използват Java като основен език за разработка на своите продукти. Сред "големите" Java не се ползва единствено от Microsoft и Apple, тъй като те разработват и поддържат собствени платформи, подобни на Java платформата.

Езикът Java е първоначално разработен и поддържан от Sun Microsystems, но през 2006 г. Java платформата става **проект с отворен код**. По-късно, през 2010 г. **Oracle купуват Sun** и в момента Java платформата се поддържа и развива от световната Java общност начело с Oracle. Благодарение на отворения код популярността на Java постоянно се увеличава и броят Java разработчици непрекъснато расте.

Езикът Java се разпространява заедно с **платформата Java** – специална среда, върху която се изпълнява Java кодът, наречена **Java Runtime Environment (JRE)**. Тази среда включва т. нар. **Java виртуална машина (JVM)** и пакет стандартни библиотеки, предоставящи базова функционалност за Java разработчиците. Благодарение на нея **Java програмите са преносими** и след като веднъж бъдат написани, след това почти без промени могат да работят на стотици хардуерни платформи и операционни системи.

Java платформата

Езикът Java не се разпространява самостоятелно, а е част от **платформата Java**. Java платформата най-общо представлява среда за разработка и изпълнение на програми, написани на езика Java. Тя се състои от езика Java, **виртуалната машина на Java (JVM)**, която изпълнява Java програмите и от съвкупност от стандартни библиотеки и инструменти за разработка, като например компилаторът, който превръща Java програмите в разбираем за виртуалната машина междинен код (Java bytecode).

За изпълнението на програми, написани на Java, е необходимо да имате инсталиран т. нар. **Java Runtime Environment (JRE)**. Това е специален софтуер, част от Java платформата, който съдържа виртуалната машина и стандартните Java библиотеки и се поддържа за различни хардуерни платформи и операционни системи. JRE е софтуер, който крайният потребител инсталира еднократно върху компютъра си, за да може да работи с Java. JRE не е стандартна част от Windows и Linux и **трябва да се инсталира допълнително**, точно както "Adobe Flash Player" се инсталира допълнително във вашия уеб браузър, за да отваряте уеб сайтове с Flash.

За разработката на Java приложения е необходимо да имате инсталиран **Java Development Kit (JDK)**. Това е пакет инструменти, с които вие като програмисти можете да пишете Java програми, да ги компилирате и изпълнявате. JDK не е необходим на крайния потребител, а само на Java разработчиците. Крайните потребители ползват JRE.

Защо Java?

Има много причини да изберем езика Java за нашата книга. Той е **съвременен език за програмиране**, **широкоразпространен**, **използван от милиони програмисти**. Същевременно Java е изключително прост и лесен за научаване език (за разлика от C и C++). Нормално е да започнем от език, който е подходящ за начинаещи и се ползва много в практиката. Именно такъв език избрахме – **лесен и много популярен**, език, който се ползва широко в индустрията от големи и сериозни фирми.

Java е обектно-ориентиран език за програмиране. Такива са всички съвременни езици, на които се разработват сериозни софтуерни системи. За предимствата на обектно-ориентираното програмиране (**ООП**) ще отделим време на много места в книгата, но за момента може да си представяте обектно-ориентираните езици като езици, които **позволяват да работите с обекти от реалния свят** (примерно човек, училище, учебник и други). Обектите имат характеристики (примерно име, цвят и т.н.) и могат да извършват действия (примерно да се движат и да говорят).

Започвайки с програмирането от езика и платформата Java вие поемате по един много перспективен път. Ако отворите някой сайт с обяви за работа за програмисти, ще се убедите, **че търсенето на Java специалисти е огромно**.

За добрия програмист **езикът, на който пише, няма съществено значение**, защото той умее да програмира. Каквито и езици и технологии да му трябват, той бързо ги овладява. Нашата цел е **не да ви научим на Java, а да ви научим на програмиране**! След като овладеете основите на програмирането и се научите да мислите алгоритмично, можете да научите и други езици и ще се убедите колко много приличат те на Java, тъй като програмирането се гради на принципи, които почти не се променят с годините и тази книга ви учи точно на тези принципи.

Примерите са върху Java 6 и Eclipse 3.4

Всички примери в книгата се отнасят за **версия 6 на езика и платформата Java**, която към момента на публикуване на книгата (декември, 2008 г.) е последната. Всички примери за използване на средата за разработка **Eclipse** се отнасят за версия **3.4** на продукта, която също е последна към момента на публикуване на книгата.

Разбира се, знанията, които придобивате за работа с **Java 6 и Eclipse 3.4**, ще можете да прилагате и за програмиране с други версии на Java и Eclipse, както и за работа с други езици за програмиране и други среди за разработка, защото всички те си приличат малко или много. **Важното е да се научите да програмирате**!

Обръщаме внимание, че настоящото издание на книгата (от април 2014 г.) внася само козметични промени в съдържанието. Книгата **не е обновена за Java 8** и новостите в езика и платформата (като ламбда изрази).

Как да четем тази книга?

Четенето на тази книга трябва да бъде съпроводено с **много, много практика**. Няма да се научите да програмирате, ако не го правите! Все едно да се научите да плувате от книга, без да пробвате. Няма начин! Колкото повече **пишете по задачите след всяка глава**, толкова повече ще научите от книгата.

Всичко, което прочетете тук, трябва **да изпробвате сами на компютъра**. Иначе няма да научите нищо. Примерно, когато прочетете за Eclipse и как да си направите първата проста програмка, трябва непременно да си изтеглите и инсталирате Eclipse и да пробвате да си направите някаква програмка. Иначе няма да се научите! На теория винаги е по-лесно, но **програмирането е практика**. Запомнете това и **правете упражненията** от книгата. Те са внимателно подбрани – хем не са много трудни, за да не ви откажат, хем не са много лесни, за да ви мотивират да приемете решаването им като предизвикателство.

Четенето на тази книга без практика е безсмислено! Трябва да отделите за писане на програми много повече време, отколкото отделяте да четете текста.

Всеки е учил математика в училище и знае, че за да се научи да решава задачи по математика, му трябва **много практика**. Колкото и да гледа и да слуша учителя, без да седне да решава задачи никой не може да се научи. Така е и с програмирането. **Трябва ви много практика**. Трябва да пишете много, да решавате задачи, да експериментирате, да се мъчите и да се борите с проблемите. Само така ще напреднете.

Не пропускайте упражненията!

На края на всяка глава има сериозен списък със **задачи за упражнения**. Не ги пропускайте! Без упражненията нищо няма да научите. След като прочетете дадена глава, трябва да седнете на компютъра и **да пробвате примерите**, които сте видели в книгата. След това трябва да се хванете и **да решите всички задачи**. Ако не можете да решите всички задачи, трябва поне да се помъчите да го направите. Ако нямате време, трябва да решите **поне първите няколко задачи от всяка глава**. Не преминавайте напред, без да решавате задачите след всяка глава! Просто няма смисъл. Задачите са малки реални ситуации, в които прилагате прочетеното. В практиката, един ден, когато станете програмисти, ще решавате всеки ден подобни задачи, но по-големи и по-сложни.

Непременно решавайте задачите за упражнения след всяка глава от книгата! Иначе рискувате нищо да не научите и просто да си загубите времето.

Колко време ще ни трябва за тази книга?

Усвояването на основите на програмирането е много сериозна задача и отнема много време. Дори и силно да ви се отдава, няма начин да се научите да програмирате на добро ниво за седмица или две, освен, ако нямате много стабилни предварителни знания и умения.

Ако искате да прочетете, разберете, научите и усвоите цялостно и в дълбочина целия учебния материал от тази книга, ще трябва да инвестирате **поне 3 месеца целодневно или поне 5-6 месеца**, ако четете и се упражнявате по малко всеки ден. Това е минималното време, за което можете **да усвоите в дълбочина основните на програмирането**.

Основният учебен материал в книгата е изложен в около **800 страници**, за които ще ви трябват около месец (по цял ден), за да го прочетете внимателно и да изпробвате примерните програми. Разбира се, трябва да отделите достатъчно внимание и на упражненията (поне двойно на това време), защото без тях почти нищо няма да научите.

Упражненията съдържат около **220 задачи** с различна трудност. За някои от тях ще ви трябват по няколко минути, докато за други ще ви трябват по няколко часа (ако въобще успеете да ги решите без чужда помощ). Това означава, че ще ви трябват **1-2 месеца по цял ден да се упражнявате** или да го правите по малко в продължение на няколко месеца.

Ако не разполагате с толкова време, замислете се **дали наистина искате да се занимавате с програмиране**. Това е много сериозно начинание, в което трябва да вложите наистина много усилия. Ако наистина искате да се научите да програмирате на добро ниво, планувайте си достатъчно време и следвайте книгата.

Ако имате въпроси по учебния материал, може да разчитате на бърз и компетентен отговор, ако зададете въпроса си във **форума на софтуерния университет**, където хиляди ваши колеги учат същия учебен материал: **https://softuni.bg/forum/**.

Защо фокусът е върху структурите от данни и алгоритмите?

Настоящата книга наред с основните познания по програмиране ви учи и на **правилно алгоритмично мислене** и работа с основните структури от данни в програмирането. **Структурите от данни и алгоритмите** са най-важните фундаментални знания на един програмист! Ако ги овладеете добре, след това няма да имате никакви проблеми да овладеете която и да е софтуерна технология, библиотека, framework или API. Именно на това разчитат и най-сериозните софтуерни фирми в света, когато наемат служители.

Интервютата за работа в Google

На интервютата за работа като софтуерен инженер в **Google** в Цюрих **100% от въпросите са върху структури от данни, алгоритми и алгоритмично мислене**. На такова интервю могат да ви накарат да реализирате на бяла дъска свързан списък (вж. главата "Линейни структури от данни") или да измислите алгоритъм за запълване на растерен многоъгълник (зададен като GIF изображение) с даден цвят (вж. метод на вълната в главата "Дървета и графи"). Изглежда Google ги интересува да наемат хора, които **имат алгоритмично мислене** и владеят основните структури от данни и базовите компютърни алгоритми. Всички технологии, които избраните кандидати ще използват след това в работата си, могат бързо да бъдат усвоени. Разбира се, не си мислете, че тази книга ще ви даде всички знания и умения, за да преминете успешно интервю за работа с Google. Знанията от книгата са абсолютно необходими, но не са достатъчни. Те са само първите стъпки.

Интервютата за работа в Microsoft

На интервютата за работа като софтуерен инженер в **Microsoft** в Дъблин голяма част от въпросите са съсредоточени върху **структури от данни, алгоритми и алгоритмично мислене**. Например могат да ви накарат да обърнете на обратно всички думи в даден символен низ (вж. главата "Символни низове") или да реализирате топологично сортиране в неориентиран граф (вж. главата "Дървета и графи"). За разлика от Google в Microsoft питат и за много **инженерни въпроси**, свързани със софтуерни архитектури, паралелна обработка (multithreading), писане на сигурен код, работа с много големи обеми от данни и тестване на софтуера. Настоящата книга далеч не е достатъчна, за да кандидатствате в Microsoft, но със сигурност знанията от нея могат да са ви полезни за една голяма част от въпросите.

Запишете се в Софтуерния университет (СофтУни)

Ако самостоятелната работа вкъщи не ви е достатъчна, **запишете се на курс**. Всеки знае, че когато има до себе си квалифициран преподавател, ученето е много по-ефективно. Именно такава възможност ви дава **Софтуерният университет (СофтУни) – https://softuni.bg**.

Софтуерният университет (СофтУни) предоставя **качествено образование, професия и работа за софтуерни инженери, с много, много практика**. Можете да тествате безплатно дали програмирането ви харесва и ви се отдава чрез безплатните курсове за начинаещи. Университетът предоставя **задълбочени знания и сериозни практически умения**, които се натрупват за две години обучение. Изучават се както основите на програмирането и компютърните науки, така и софтуерните технологии, които се търсят на пазара на труда **– бази данни**, **уеб разработка**, **мобилни приложения** и други.

След първата година болшинството от **студентите започват работа**, след препоръка от СофтУни към партньорска фирма, а втората година изкарват паралелно докато **учат и работят**. По желание се учи още 2 години за държавно-призната диплома за висше образование във ВУЗ-партньор.

Научете повече за СофтУни от неговия сайт: http://softuni.bg.

Поглед към съдържанието на книгата

Нека сега разгледаме накратко какво ни предстои в **следващите глави** на книгата. Ще разкажем по няколко изречения за всяка от тях, за да знаете какво ви очаква да научите.

Глава 1. Въведение в програмирането

В главата "Въведение в програмирането" ще разгледаме основните термини от програмирането и **ще напишем първата си програма**. Ще се запознаем с това какво е програмиране и каква е връзката му с компютрите и програмните езици. Накратко ще разгледаме основните етапи при писането на софтуер. Ще въведем **езика Java** и ще се запознаем с Java платформата и Java технологиите. Ще разгледаме какви помощни средства са ни необходими, за да можем да програмираме на Java. Ще използваме Java, за да **напишем първата си програма**, ще я **компилираме** и **изпълним** както от командния ред, така и от среда за разработка Eclipse. Ще се запознаем с документацията на Java, която позволява по-нататъшно изследване на възможностите на езика.

Глава 2. Примитивни типове и променливи

В главата "Примитивни типове и променливи" ще разгледаме **примитивните типове и променливи в Java** – какво представляват и как се работи с тях. Първо ще се спрем на типовете данни – целочислени типове, реални типове с плаваща запетая, булев тип, символен тип, обектен тип и стрингов тип. Ще продължим с това какво е **променлива**, какви са нейните характеристики, как се декларира, как се присвоява стойност и какво е инициализация на променлива. Ще се запознаем и с друго наименование на променливите, а по-точно – "**идентификатор**". Към края на главата ще се спрем на литералите.

Глава 3. Оператори и изрази

В главата "Оператори, изрази" ще се запознаем с **операторите и действията**, които те извършват върху различните типове данни. Ще разясним приоритета на операторите и групите оператори според броя на аргументите, които приемат и това какво действие извършват. Ще разгледаме **аритметичните**, **битовите** и **булевите** оператори. След това ще разгледаме **преобразуването на типове**, защо е нужно и как да се работим с него. Накрая ще разясним какво представляват **изразите** и как се използват в програмирането.

Глава 4. Вход и изход от конзолата

В главата "Вход и изход от конзолата" ще се запознаем с **конзолата**. Ще обясним какво представлява тя, кога и как се използва, какви са принципите на повечето програмни езици за достъп до конзолата. Ще се запознаем с някой от възможностите на Java за взаимодействие с потребителя. Ще разгледаме основните **потоци за входно-изходни операции**: `System.in`, `System.out` и `System.err`, класът `Scanner` и използването на форматиращи низове за отпечатване на данни в различни формати.

Глава 5. Условни конструкции

В главата "Условни конструкции" ще разгледаме **условните конструкции** в Java, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Ще обясним синтаксиса на **условните оператори**: `if` и `if-else` с подходящи примери и ще разясним практическото приложение на **оператора за избор** `switch`. Ще обърнем внимание на добрите практики, които е нужно да бъдат следвани, с цел постигане на по-добър стил на програмиране при използването на **вложени условни конструкции**.

Глава 6. Цикли

В главата "Цикли" ще разгледаме **конструкциите за цикли**, с които можем да изпълняваме даден фрагмент програмен код многократно. Ще разгледаме как се реализират **повторения с условие** (`while` и `do-while` цикли) и как се работи с **`for`-цикли**. Ще дадем примери за различните възможности за дефиниране на цикъл, за начина им на конструиране и за някои от основните им приложения. Накрая ще разгледаме как можем да използваме няколко цикъла един в друг (**вложени цикли**).

Глава 7. Масиви

В главата "Масиви" ще се запознаем с **масивите** като средства за обработка на **поредица от еднакви по тип елементи**. Ще обясним какво представляват масивите, как можем да декларираме, създаваме и инициализираме масиви. Ще обърнем внимание на **едномерните** и **многомерните масиви**. Ще разгледаме различни начини за обхождане на масив, четене от стандартния вход и отпечатване на стандартния изход. Ще дадем много примери за задачи, които се решават с използването на масиви и ще ви покажем колко полезни са те.

Глава 8. Бройни системи

В главата "Бройни системи" ще разгледаме начините на работата с **различни бройни системи** и представянето на числата в тях. Повече внимание ще отделим на представянето на числата в **десетична**, **двоична** и **шестнадесетична бройна система**, тъй като те се използват масово в компютърната техника и в програмирането. Ще обясним и начините за

кодиране на числовите данни в компютъра и видовете кодове, а именно: прав код, обратен код, допълнителен код и двоично-десетичен код.

Глава 9. Методи

В главата "Методи" ще се запознаем подробно с това какво е **метод** и защо се използват методи. Ще разберем как се декларират методи и какво е сигнатура на метод. Ще научим **как да създадем собствен метод** и съответно как да го използваме (**извикваме**). Ще разберем как можем да използваме **параметри** в методи и как **да върнем резултат** от метод. Накрая ще препоръчаме някои утвърдени практики при работата с методи.

Глава 10. Рекурсия

В главата "Рекурсия" ще се запознаем с **рекурсията** и нейните приложения. Рекурсията представлява мощна техника, при която един метод извиква сам себе си. С нея могат да се решават сложни **комбинаторни задачи**, при които с лекота могат да бъдат изчерпвани различни комбинаторни конфигурации. Ще ви покажем много примери за правилно и неправилно използване на рекурсия и ще ви убедим колко полезна може да е тя.

Глава 11. Създаване и използване на обекти

В главата "Създаване и използване на обекти" ще се запознаем накратко с основните понятия в обектно-ориентираното програмиране – **класовете** и **обектите** – и ще обясним как да използваме класовете от стандартните библиотеки на Java. Ще се спрем на някои често използвани системни класове и ще видим как се създават и използват техни **инстанции** (обекти). Ще разгледаме как можем да осъществяваме достъп до **полетата** на даден обект, как да извикваме **конструктори** и как да работим със **статичните полета** в класовете. Накрая ще се запознаем с понятието **пакети** – какво ни помагат, как да ги включваме и използваме.

Глава 12. Обработка на изключения

В главата "Обработка на изключения" ще се запознаем с **изключенията** в Java и обектно-ориентираното програмиране. Ще се научим как да ги **прихващаме и обработваме** чрез **конструкцията** `try-catch`, как да ги предаваме на извикващи методи чрез `throws` и как **да хвърляме** собствени или прихванати изключения. Ще дадем редица примери за използването им. Ще разгледаме типовете изключения и **йерархията**, която образуват. Накрая ще се запознаем с предимствата при използването на изключения и с това как най-правилно да ги прилагаме в конкретни ситуации.

Глава 13. Символни низове

В главата "Символни низове" ще се запознаем със **символните низове**: как са реализирани те в Java и по какъв начин можем да **обработваме**

текстово съдържание. Ще прегледаме различни методи за манипулация на текст; ще научим как да извличаме поднизове по зададени параметри, как да търсим за ключови думи, както и да отделяме един низ по разделители. Ще се запознаем с методи и класове за по-елегантно и стриктно **форматиране на текстовото съдържание** на конзолата, с различни методики за извеждане на числа, а също и с извеждането на отделни компоненти на текущата дата. Накрая ще предоставим полезна информация за **регулярните изрази** и ще научим по какъв начин да извличаме данни, отговарящи на определен шаблон.

Глава 14. Дефиниране на класове

В главата "Дефиниране на класове" ще разберем как можем **да дефинираме собствени класове** и кои са елементите на класовете. Ще се научим да декларираме **полета**, **конструктори**, **свойства** и **методи** в класовете. Ще припомним какво е метод и ще разширим знанията си за **модификатори** и нива на достъп до полетата и методите на класовете. Ще разгледаме особеностите на конструкторите и подробно ще обясним как обектите се съхраняват в динамичната памет и как се инициализират полетата им. Накрая ще обясним какво представляват **статичните елементи** на класа – полета (включително константи), свойства и методи и как да ги ползваме.

Глава 15. Текстови файлове

В главата "Текстови файлове" ще се запознаем с основните похвати при **работа с текстови файлове** в Java. Ще разясним какво е това **поток**, за какво служи и как се ползва. Ще обясним какво е **текстов файл** и как се чете и пише в текстови файлове. Ще демонстрираме и обясним добрите практики за прихващане и **обработка на изключения**, възникващи при работата с файлове. Разбира се, всичко това ще онагледим и демонстрираме на практика с **много примери**.

Глава 16. Линейни структури от данни

В главата "Линейни структури от данни" ще се запознаем с **абстрактните структури данни** в програмирането. Ще разгледаме как при определена задача една структура е по-ефективна и удобна от друга. Ще разгледаме структурите "**списък**", "**стек**" и "**опашка**" и техните разновидности, начини за тяхната имплементация и приложение в практиката. Ще се научим да ползваме готови Java класове като `ArrayList<T>` за имплементация на списъчни структури.

Глава 17. Дървета и графи

В главата "Дървета и графи" ще разгледаме т. нар. **дървовидни структури от данни**, каквито са **дърветата** и **графите**. Познаването на тези структури е важно за съвременното програмиране. Всяка от тях се използва за моделирането на проблеми от реалността, които се решават ефективно с тяхна помощ. Ще разгледаме в детайли какво представляват дървовидните

структури данни и ще покажем техните основни предимства и недостатъци. Ще дадем **примерни реализации** и задачи, демонстриращи реалната им употреба. Ще се спрем по-подробно на **двоичните дървета**, **наредените двоични дървета за претърсване** и **балансираните дървета**. Ще разгледаме структурата от данни "**граф**", видовете графи и тяхната употреба. Ще покажем как се работи с вградените в Java платформата имплементации на **балансирани дървета**.

Глава 18. Речници, хеш-таблици и множества

В главата "Речници, хеш-таблици и множества" ще разгледаме някои по-сложни структури от данни като **речници** и **множества**, и техните реализации с **хеш-таблици** и **балансирани дървета**. Ще обясним в детайли какво представляват **хеширането** и хеш-таблиците и защо са толкова важни в програмирането. Ще дискутираме понятието "**колизия**" и как се получават колизиите при реализация на хеш-таблици и ще предложим различни подходи за разрешаването им. Ще разгледаме абстрактната структура данни "**множество**" и ще обясним как може да се реализира чрез **речник** и чрез **балансирано дърво**. Ще дадем примери, които илюстрират приложението на описаните структури от данни в практиката.

Глава 19. Структури от данни – съпоставка и препоръки

В главата "Структури от данни – съпоставка и препоръки" ще съпоставим една с друга структурите данни, които разгледахме до момента, по отношение на **скоростта**, с която извършват основните операции (добавяне, търсене, изтриване и т.н.). Ще дадем конкретни **препоръки в какви ситуации какви структури от данни да ползваме**. Ще обясним кога да предпочетем хеш-таблица, кога масив, кога динамичен масив, кога множество, реализирано чрез хеш-таблица и кога балансирано дърво. Всички тези структури имат вградена имплементация в Java платформата. От нас се иска единствено да можем **да преценяваме кога коя структура да ползваме**, за да пишем ефективен и надежден програмен код. Именно на това е посветена тази глава – на **ефективната работа** със структури от данни в Java.

Глава 20. Принципи на обектно-ориентираното програмиране

В главата "Принципи на обектно-ориентираното програмиране" ще се запознаем с **принципите на обектно-ориентираното програмиране**: **наследяване** на класове и имплементиране на **интерфейси**, **абстракция** на данните и поведението, **капсулация** на данните и скриване на информация за имплементацията на класовете, **полиморфизъм** и **виртуални методи**. Ще обясним в детайли принципите за свързаност на отговорностите и функционално обвързване (**cohesion** и **coupling**). Ще опишем накратко как се извършва обектно-ориентирано моделиране и как се

създава обектен модел по описание на даден бизнес проблем. Ще се запознаем с **езика UML** и ролята му в процеса на обектно-ориентираното моделиране. Накрая ще разгледаме съвсем накратко концепцията "**шаблони за дизайн**" и ще дадем няколко типични примера за шаблони, широко използвани в практиката.

Глава 21. Качествен програмен код

В главата "Качествен програмен код" ще разгледаме основните правила за **писане на качествен програмен код**. Ще бъде обърнато внимание на **именуването** на елементите от програмата (променливи, методи, класове и други), правилата за **форматиране** и подреждане на кода, добрите практики за изграждане на **висококачествени методи** и принципите за **качествена документация** на кода. Ще бъдат дадени много примери за качествен и некачествен код. Ще бъдат описани и официалните **код конвенции** от Oracle за писане на Java, както и JavaBeans спецификацията. В процеса на работа ще бъде обяснено как да се използва средата за програмиране Eclipse, за да се автоматизират някои операции като **форматиране** и **преработка на кода**.

Глава 22. Как да решаваме задачи по програмиране?

В главата "Как да решаваме задачи по програмиране?" ще дискутираме един **препоръчителен подход за решаване на задачи по програмиране** и ще го илюстрираме нагледно с реални примери. Ще дискутираме **инженерните принципи**, които трябва да следваме при решаването на задачи (които важат в голяма степен и за задачи по математика, физика и други дисциплини) и ще ги покажем в действие. Ще опишем **стъпките**, през които преминаваме при решаването на няколко примерни задачи и ще демонстрираме какви грешки се получават, ако не следваме тези стъпки. Ще обърнем внимание на някои важни стъпки от решаването на задачи (като например **тестване**), които обикновено се пропускат.

Глави 23, 24, 25. Примерни теми от изпити по програмиране

В главите "Примерни теми от изпити по програмиране" ще разгледаме условията и ще предложим решения на девет примерни задачи от три примерни изпита по програмиране, проведени на 30.09.2005 г., на 8.04.2006 г. и на 11.12.2005 г. При решаването им ще приложим на практика описаната методология в темата "Как да решаваме задачи по програмиране".

За използваната терминология

Тъй като настоящият текст е на български език, ще се опитаме да ограничим употребата на **английски термини**, доколкото е възможно. Съществуват обаче основателни причини да използваме и английските термини наред с българските им еквиваленти:

- По-голямата част от техническата документация за Java е на **английски език** (повечето книги и в частност официалната документация) и затова е много важно читателите да знаят английския еквивалент на всеки използван **термин**.
- Много от използваните термини не са пряко свързани с Java и са навлезли отдавна в **програмисткия жаргон** от английски език (например "дебъгвам", "компилирам" и "плъгин"). Тези термини ще бъдат изписвани най-често на кирилица.
- Някои термини (например "framework" и "deployment") са **трудно преводими** и трябва да се използват **заедно с оригинала в скобки**. В настоящата книга на места такива термини са превеждани по различни начини (според контекста), но винаги при първо срещане се дава и оригиналният термин на английски език.

Как възникна тази книга?

Често се случва някой да ме попита **от коя книга да започне да се учи на програмиране**. Срещат се ентусиазирани младежи, които искат да се учат да програмират, но не знаят от къде да започнат. За съжаление и аз не знам коя книга да им препоръчам. Сещам се за много книги за Java – и на български и на английски, но **никоя от тях не учи на програмиране**. Няма много книги (особено на български език), които да учат на концепциите на програмирането, на **алгоритмично мислене**, на **структури от данни**. Има книги за начинаещи, които учат на езика Java, но не и на основите на програмирането. Има и няколко хубави книги за програмиране на български език, но са вече остарели. Знам няколко такива книги за C и Паскал, но не и за Java или C#. В крайна сметка **до скоро нямаше хубава книга за програмиране**, която горещо да препоръчам на всеки, който иска да се захване с програмиране от нулата.

Липсата на хубава книга по програмиране за начинаещи в един момент ни мотивира **да се хванем и да напишем такава книга**. Решихме, че можем да помогнем и да дадем знания и вдъхновение на много млади хора да се захванат сериозно с програмиране.

Историята на тази книга

Историята на тази книга е дълга и интересна. Тя започва с въведителните курсовете по програмиране в Национална академия по разработка на софтуер (НАРС) през 2005 г., когато под ръководството на Светлин Наков за тях е изготвено учебно съдържание за **курс "Въведение в**

програмирането със C#". След това то е адаптирано към Java и така се получава **курсът "Въведение в програмирането с Java"**. През годините това учебно съдържание претърпява доста промени и подобрения първо в Софтуерната академия на Телерик, а след това в Софтуерния университет и достига до един изчистен и завършен вид.

Към **стартирането на проекта** (август 2008) курсът "Въведение в програмирането с Java" се преподава в Националната академия по разработка на софтуер по учебни материали с обем 720 PowerPoint слайда, заедно с около 110 демонстрационни примера и над 200 задачи за упражнения, разделени в 15 отделни теми. Това е учебен материал по основи на програмирането, който съдържа **най-важните концепции**, с които един начинаещ трябва да стартира. Това е материал, който преработван и редактиран десетки пъти, добил вече зрялост, добре систематизиран и с проверена ефективност. Защо да не го ползваме като **основа за съдържанието на книгата**? Ние точно това направихме – решихме да напишем книгата ръководейки се от учебните материали, които бяхме разработили за нашите курсове по програмиране, като добавим от себе си нашия опит и допълним с още информация, примери и интересни задачи.

Събиране на авторския екип

Работата по книгата започва в един топъл летен ден, когато **основният автор Светлин Наков**, вдъхновен от идеята за написване на учебник за курсовете по "Въведение в програмирането" събира **екип от двадесетина млади софтуерни инженери**, ентусиасти, които имат желание да споделят знанията си и да напишат по една глава от книгата.

Светлин Наков дефинира **учебното съдържание** и го разделя в глави и създава шаблон за съдържанието на всяка глава. Шаблонът съдържа структурата на текста – всички основни заглавия в дадената глава и всички подзаглавия. Остава да се напишат текста, примерите и задачите.

На първата **среща на екипа** учебното съдържание претърпява малко промени. По-обемните глави се разделят на няколко отделни части (например структурите от данни), възникват няколко нови глави (например работа с изключения) и се определят автори и редактори за всяка глава. Идеята е проста: **всеки да напише по 1 глава от книгата** и накрая да ги съединим. За да няма голяма разлика в стиловете, форматирането и начина на представяне на информацията авторите приемат единно ръководство на писателя, в което строго се описват всички правила за писане. В крайна сметка всеки си има тема и писането започва.

За проекта се създава **сайт за съвместна работа в екип** в Google Code, който е по-късно мигриран към GitHub. Там стоят последните версии на всички текстове и материали по настоящата безплатна книга за програмиране: https://github.com/nakov/introjavabook.

Задачите и сроковете

Както във всеки проект, след **разпределяне на задачите** се **слагат крайни срокове** за всяка от тях, за да се планира работата във времето. По план книгата трябва да излезе от печат през октомври, но това не се случва в срок, защото много от авторите се забавят, а някои въобще не изпълняват поетия ангажимент.

Когато идва първия краен срок едва половината от авторите са готови на време. **Сроковете се удължават** и голяма част от авторите завършват работата по своята глава. Започва **работата на редакторите**. Паралелно някои автори дописват. За някои глави се търсят нови автори, защото оригиналният автор се проваля и бива отстранен.

Няколко месеца по-късно **книгата е готова на 90%**, авторите загубват ентусиазъм работата започва да върви много бавно и мъчно. Светлин Наков се опитва да компенсира и да **дописва недовършените теми**, но работата е много. Въпреки труда, който той влага всеки свободен уикенд, работата е много и все не свършва месеци наред.

Всички автори подценяват сериозно обема на работата и това е основно причината за забавянето на поява на книгата. Авторите си мислят, че писането става бързо, но истината е, че **за една страница текст** (четене, писане, редактиране, преправяне и т.н.) **отива средно по 1 час работа**, та дори и повече. Сумарно за написването на цялата книга **са вложени около 800-1000 работни часа труд**, разпределени сред всички автори и редактори, което се равнява на над 6 месеца работа на един автор на пълен работен ден. Понеже всички автори пишеха в свободното си време, работата вървеше бавно и отне 4-5 месеца.

Авторският колектив

Авторският колектив е наистина главният виновник за съществуването на тази книга. Написването на текст с такъв обем и такова качество е сериозна задача, която изисква много време.

Идеята за участие на **толкова много автори** е добре проверена, тъй като по подобен начин са написани вече няколко други книги (като "Програмиране за .NET Framework"). Въпреки, че отделните глави от книгата са писани от **различни автори**, те следват **единен стил и високо качество**. Всички глави са добре структурирани, с много заглавия и подзаглавия, с много и подходящи примери, с добър стил на изказ и еднакво форматиране.

Екипът, написал настоящата книга, е съставен от хора, които имат силен интерес към програмирането и желаят б**езвъзмездно да споделят своите знания** като участват в написването на една или няколко от темите. Някои от участниците в екипа са бивши **студенти на Наков**, преминали успешно неговите обучения. Други са щатни преподаватели по програмиране. Трети са просто ентусиасти. Най-хубавото е, че всички автори, съавтори и редактори от екипа по разработката на книгата са п**рограмисти с реален**

практически опит, което означава, че читателят ще почерпи знания, практики и съвети от хора, реализирали се в софтуерната индустрия.

Участниците в проекта дадоха своя труд безвъзмездно, без да получат материални или други облаги, защото подкрепяха идеята за написване на добра книга за начинаещи програмисти на български език и имаха силно желание да помогнат на своите бъдещи колеги да навлязат бързо в програмирането.

Следва кратко **представяне на авторите** (по азбучен ред, към лятото на 2008 г.). Най-вероятно информацията за тях вече е остаряла.

Борис Вълков

Борис Вълков е софтуерен инженер във фирма CSC (www.csc.com). Той е преподавател във Факултета по математика и информатика на Пловдивски университет "Паисий Хилендарски", където е завършил своите бакалавърска и магистърска степени. Има опит с **разработката на висококачествен софтуер** в областта на застраховането и здравеопазването. Борис е сертифициран разработчик, използвал широк спектърът от технологии, предимно свързани със C/C++ и Java. Неговите професионални интереси включват софтуерни архитектури, дизайн и процеси за ефективно управление на софтуерни проекти. Можете да се свържете с него по e-mail: b.valkov@gmail.com.

Веселин Колев

Веселин Колев е старши софтуерен инженер, ръководил различни екипи и проекти. Като програмист има опит с разнообразни технологии, част от които са **C/C++, .NET и Java**. Проектите, по които е работил, включват големи уеб базирани системи, системи за машинен превод, икономически софтуер и др. Интересите му обхващат разработването на **съвременни приложения от всякакъв характер**, в които се държи на **качеството**. Веселин има опит като лектор в Националната академия по разработка на софтуер в областта на .NET технологиите. В момента той следва Компютърни науки във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Можете да се свържете с него по e-mail: vesko.kolev@gmail.com. Неговият личен блог е достъпен от адрес: http://veskokolev.blogspot.com.

Владимир Цанев (Tsachev)

Владимир Цанев е софтуерен разработчик в Национална академия по разработка на софтуер (НАРС). Интересите му са свързани най-вече с open-source технологиите, използвани с **Java платформата**. Участвал е в разработката на проекти в областта на Интернет услугите, големи и малки информационни системи за управление и други.

Владимир е лектор в Националната академия по разработка на софтуер (НАРС), където **води различни курсове** свързани с разработката на софтуерни продукти с Java.

Завършил е бакалавърска степен по специалност "Приложна математика" във Факултета по математика и информатика на Софийски Университет "Св. Климент Охридски". По време на следването си там води упражнения по **програмиране с Java**. В момента следва магистратура със специалност "Софтуерни технологии" в същия факултет.

Личната му уеб страница се намира на адрес http://info.tsachev.org.

Данаил Алексиев

Данаил Алексиев е студент в трети курс в Техническия Университет – София, специалност "Компютърни системи и технологии". Работи като програмист във фирма Insight Technologies (www.insight-bg.net) и лектор към Национална академия по разработка на софтуер (НАРС). Основните му професионални интереси са насочени към **уеб програмирането**. Данаил е основен участник в **Java проектите** в учебната лаборатория Unidevelop (http://unidevelop.org) към ТУ-София. Можете да се свържете с него по e-mail: danail_al@abv.bg.

Лъчезар Божков

Лъчезар Божков е студент, последен курс, в Технически Университет – София, специалност ИИ на английски език. Той има значителен опит в работата с **.NET Framework**, Visual Studio и Microsoft SQL Server. Работил е като **програмист** и преподавател за Национална академия по разработка на софтуер (НАРС) и ИТ отдела на Кока-Кола Ботлърс, Ирландия, Дъблин. Можете да се свържете с Лъчезар по e-mail: lachezar.bozhkov@gmail.com.

Лъчезар Цеков

Лъчезар Цеков е софтуерен инженер и изследовател в областта на семантичните технологии във фирма Ontotext Lab (www.ontotext.com). Завършил е Технически Университет София специалност "Комуникационна Техника и Технологии". Лъчезар е работил в няколко фирми като **старши програмист** и има поглед над цялостния производствен цикъл на софтуерните продукти. Основната експертиза на Лъчезар е в проектирането и реализацията на големи **корпоративни приложения базирани на Java и SOA** технологиите. Лъчезар е голям привърженик на правилното използване на принципите на **обектно-ориентираното програмиране** и прилагането на подходящите **структури от данни**. За връзка можете да използвате неговия e-mail: luchesar.cekov@gmail.com.

Марин Георгиев

Марин Георгиев е **SAP консултант** в Coca Cola HBC IT Shared Services. Получава своята бакалавърска степен в Технически университет - София

след частичното си следване и разработване на дипломната си работа в Италия, в Universita degli Studi di Pavia. Завършил е Националната академия за разработване на софтуер (НАРС) и е сертифициран като **Sun Certified Java Programmer**. Професионалните интереси на Марин са насочени към уеб-базираните Java технологии – EJB, Seam, JSF и внедряване на SAP ERP решения. Можете да се свържете с Марин Георгиев на e-mail: adalmirant@abv.bg.

Марио Пешев

Марио Пешев е софтуерен разработчик във фирма Nonillion (www.nonillion.com), **Sun сертифициран Java инженер**. Той участва активно в писането на технически статии и материали, разработка на външни проекти като **freelance developer** и водене на курсове по програмиране с Java. В момента Марио следва Информатика в Нов български университет. Професионалните му интереси включват работа с иновативни технологии, информационна сигурност, софтуерни архитектури и дизайн. Марио е огромен фен на платформата за управление на уеб съдържание **WordPress**. Личният му блог е http://peshev.net/blog.

Мариян Ненчев

Мариян Ненчев е софтуерен разработчик в Insight Technologies (www.insight-bg.net) и лектор в Национална академия по разработка на софтуер (academy.devbg.org). Водил е редица курсове за обучение и преквалифициране на софтуерни специалисти в ИТ индустрията. Той е студент, последен курс, във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност информатика. Професионалните му интереси са насочени към разработката на **enterprise приложения с Java технологиите**, .NET платформата и информационната сигурност в Web среда. Можете да се свържете с Мариян по e-mail: nenchev.mariyan@gmail.com.

Михаил Стойнов

Михаил Стойнов е магистър "Стопанско Управление" в Софийски Университет. Той е завършил е Информатика отново в Софийски Университет. В момента е **софтуерен разработчик** в голяма българска компания.

От няколко години Михаил е **хоноруван преподавател** във Факултета по математика и информатика като досега е водил част от лекциите на "**Съвременни Java технологии**", "**Програмиране за .NET Framework**", "**Разработка на Java уеб приложения**", "**Шаблони за дизайн**" и "**Качествен програмен код**". Активно е участвал в създаването на учебните материали за курсовете. Водил е софтуерен курс в Нов Български Университет.

Михаил е участвал като **съавтор в книгата "Програмиране за .NET Framework"**. Той е водил множество лекции свързани с Java и .NET

Framework на различни конференции и семинари. Участвал е като лектор в академичните дни на Майкрософт.

Михаил е водил **IT обучения** в няколко компании в България и една в чужбина. Бил е лектор на курсове по Java, Java EE, SOA, Spring в Националната академия по разработка на софтуер (НАРС).

Михаил е работил в офисите на компании като Siemens, HP, EDS в Холандия, Германия и други държави, като там е натрупал сериозен опит както за софтуерното изкуство, така и за качественото писане на софтуер чрез участието си в големи софтуерни проекти. Занимава се усилено с **информационна сигурност** и одити за сигурността на софтуерни системи.

Личният му блог е достъпен на адрес: http://mihail.stoynov.com/blog/.

Николай Василев

Николай Василев е завършил бакалавърската си степен във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност "Математика и информатика". В момента е студент към магистърските програми на Софийски университет "Св. Климент Охридски", специалност "Уравнения на математическата физика и приложения" и университета в Малага, Испания, специалност "**Софтуерно инженерство и изкуствен интелект**".

По време на създаването на книгата има четиригодишен опит като **програмист в различни софтуерни компании**.

Той е **сертифициран Sun програмист** за Java 2 Platform, SE 5.0.

В периода 2002-2005 г е бил асистент към курсовете по програмиране водени от доц. Божидар Сендов, "**Увод в програмирането**" и "**Структури от данни и програмиране**".

Интересите му са свързани с **проектирането и имплементацията на софтуер** и участие в академични дейности.

За връзка с него можете да използвате неговата електронна поща: nikolay.vasilev@gmail.com.

Николай Недялков

Николай Недялков е президент на Асоциация за информационна сигурност, **технически директор** на портала за електронни разплащания и услуги eBG.bg и **бизнес консултант** в други компании. Николай е **професионален разработчик на софтуер**, консултант и преподавател с дългогодишен опит. Той е автор на редица статии и публикации и лектор на множество конференции и семинари в областта на софтуерните технологии и **информационната сигурност**. Преподавателският му опит се простира от асистент по "Структури от данни в програмирането", "Обектно-ориентирано програмиране със C++" и "Visual C++" до лектор в курсовете "Мрежова сигурност", "Сигурен програмен код", "Интернет

програмиране с Java", "Конструиране на качествен програмен код", "Програмиране за платформа .NET" и "Разработка на приложения с Java". Интересите на Николай са концентрирани върху **техническата и бизнес страната на информационната сигурност**, Java и .NET технологиите и моделирането и управлението на **бизнес процеси** в големи организации. Николай има бакалавърска и магистърска степен от Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Като ученик е **дългогодишен състезател по програмиране**, с редица призови отличия и награди. Неговият уеб сайт е на адрес: http://www.nedyalkov.com.

Петър Велев

Петър Велев е софтуерен инженер и изследовател. Завършил е Технически университет в София по специалност Информационни технологии към факултета по "Компютърни системи и управление". Петър е работил в различни фирми като младши **програмист на Java**. Понастоящем е докторант в Института по информационни технологии при БАН – София. Основните му интереси са в областта на **софтуерното инженерство** и **обектно-ориентираното програмиране**. Той проявява интерес и в областите на динамичните онтологии, многоагентните системи и структурите от данни. За връзка с Петър можете да използвате неговия e-mail: petone681@gmail.com.

Радослав Иванов

Радослав Иванов е софтуерен инженер в Европейската организация за ядрени изследвания (**CERN**) – www.cern.ch. Завършил е Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" и има сериозен **професионален опит в разработката на софтуер**. Той е лектор в редица курсове в Софийски университет "Св. Климент Охридски", частни компании и организации и е съавтор на книгата "Програмиране за .NET Framework". Сред професионалните му интереси са **Java технологиите**, **.NET платформата**, архитектура и дизайн на софтуерни системи и др.

Румяна Топалска

Румяна Топалска работи като софтуерен разработчик във фирма Fadata (www.fadata.bg). Завършила е своята бакалавърска степен в немския факултет на Техническия университет в гр. София, специалност "Компютърни системи и технологии" и в момента следва своята магистратура в същия университет. Основните професионални интереси на Румяна са в **уеб програмирането**, **уеб услугите** и **интеграцията на софтуерни системи**. Може да се свържете с нея на mail: rumchoto@abv.bg.

Стефан Стаев

Стефан Стаев е софтуерен разработчик във фирма ICB (www.icb.bg). Занимава се с разработката на **уеб базирани системи** върху .NET платформата. В момента той следва Информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Професионалните му интереси са в областта на **уеб технологиите**. Можете да се свържете с него по e-mail: s.staev@gmail.com.

Светлин Наков

Светлин Наков е вдъхновител на хиляди млади хора да се захванат с технологии и програмиране. Той е създател на **мащабни образователни инициативи** за софтуерни инженери, чрез които **дава професия и работа на хиляди млади хора**: Национална академия по разработка на софтуер, Софтуерна академия на Телерик, Софтуерен университет (СофтУни) и др.

Светлин Наков има **20+ години опит** като софтуерен инженер, преподавател, консултант, ръководител на проекти и предприемач. Разработвал е уеб базирани системи с .NET, Java, Oracle и други технологии.

В момента Светлин Наков вдъхновява и задвижва **Софтуерния университет** (**СофтУни** – **http://softuni.bg**) – най-мащабният център за практическо обучение на софтуерни инженери в България, където **обучава хиляди млади хора на програмиране**, разработка на софтуер, компютърни науки, софтуерни технологии и работа в екип и има дава успешен кариерен старт.

Наков е **автор на 7 книги за програмиране** и софтуерни технологии, C# и Java. Публикувал е десетки **научни и технически статии** и разработки. Водил е **стотици обучения** в сферата на софтуерните технологии и редовно изнася презентации на **технологични конференции и семинари** за програмиране, разработка на софтуер и образование. Има **докторска степен** (PhD) по компютърни науки, медали от международни олимпиади и състезания по програмиране и е носител **наградата "Джон Атанасов" на българския Президент** за принос към развитието на информационните технологии и информационното общество. Преподавал е в Национална академия по разработка на софтуер (НАРС), Софийски университет "Св. Климент Охридски", Технически университет – София, Нов Български университет (НБУ), Софтуерна академия на Телерик и Софтуерен университет (СофтУни).

Научете повече за Светлин Наков и проектите, с които се занимава от неговия личен сайт: http://www.nakov.com.

Теодор Стоев

Теодор Стоев е **софтуерен разработчик** във фирма Wizefish (www.wizefish.com). Завършил е специалност Информатика във Факултета по математика и информатика на Софийски университет "Св. Климент

Охридски". В момента следва магистърска програма в същия факултет, специалност Софтуерни технологии. Неговите професионални интереси са в областта на **обектно-ориентирания анализ**, **моделиране** и **изграждане на софтуерни приложения**, **уеб технологиите** и в частност изграждането на **RIA** (Rich Internet Applications). Зад гърба си има сериозен опит с алгоритмично програмиране; участвал е в редица ученически и студентски национални състезания по информатика.

Христо Тодоров

Христо Тодоров е софтуерен разработчик във фирма Fadata (www.fadata.bg). Завършил е своята бакалавърска степен във Факултета по компютърни системи и управление на ТУ-София, където следва своята магистратура. Занимава се с **разработка на уеб приложения**, базирани на Java. Неговите професионални интереси включват дизайн, разработка и защита на уеб информационни системи. Можете да се свържете с него по e-mail: hgt.todorov@gmail.com.

Цвятко Конов

Цвятко Конов е софтуерен разработчик и инструктор в Национална академия по разработка на софтуер (НАРС). Той има опит с **.NET** и **Java** технологиите. В момента следва Приложна математика в Софийски университет "Св. Климент Охридски". Професионалните му интереси включват работа с **иновативни технологии**, **софтуерни архитектури** и дизайн, както и **преподавателска дейност**. Личният му блог е на адрес http://tsvyatkokonov.blogspot.com/.

Редакторите

Освен авторите сериозен принос за създаването на книгата имат и **редакторите**, които участваха безвъзмездно в проверката на текста и примерите и **отстраняването на грешки и проблеми**. Следват редакторите и техния принос (по азбучен ред).

- **Георги Пенчев** – редактор на главата "Линейни структури от данни".
- **Ивайло Иванов** – редактор на главите "Примитивни типове и променливи", "Оператори и изрази" и "Вход и изход от конзолата".
- **Мирослав Николов** – редактор на главите "Цикли" и "Методи".
- **Николай Давков** – редактор на главата "Масиви".
- **Марио Пешев** – редактор на главите "Принципи на обектно-ориентираното програмиране".
- **Пламен Табаков** – редактор на главите "Въведение в програмирането", "Примитивни типове и променливи", "Оператори и изрази" и "Вход и изход от конзолата".
- **Радко Люцканов** – редактор на главата "Обработка на изключения".

- **Светлин Наков** – редактирал детайлно почти всички глави (с дребни изключения).
- **Тодор Балабанов** – редактор на главите "Рекурсия", "Примерна тема от изпит по програмиране – 8.04.2006".
- **Тодор Сираков** – редактор на главите "Условни конструкции" и "Символни низове".
- **Цветан Василев** – редактор на главите "Масиви", "Методи" и "Дефиниране на класове".

Дизайн на корицата: **Станимира Иванова**.

Отзиви

В тази секция сме събрали отзиви и **мнения за настоящата книга от известни български софтуерни инженери**, с дългогодишен опит в програмирането, реализирали се повече от успешно в професията си. Надяваме се, че всеки от вас, който иска да бъде успешен софтуерен инженер като тях и да програмира професионално в сериозна софтуерна компания, да послуша съветите им и да отдели достатъчно време на тази книга и на решаването на задачите в нея. Ако не вярвате на авторитетния авторски колектив, **повярвайте на колегите от Microsoft, Google, SAP AG, telerik** и всички останали.

Отзив от Николай Манчев

Това е **най-изчерпателната книга за начинаещи Java разработчици**, достъпна в България. Ако търсите реализация в тази област, не се колебайте – това е книгата, която ви е нужна!

За да станете добър разработчик на софтуер, трябва да сте готов да инвестирате в натрупването на познания в няколко области и конкретния език за програмиране е само една от тях. Добрият разработчик трябва да познава не само синтаксиса и приложно-програмния интерфейс на езика, който си е избрал. Той трябва да притежава също така **задълбочени познания по обектно-ориентирано програмиране, структури от данни и писане на качествен код**. Той трябва да подкрепи тези си познания и със сериозен практически опит.

Когато започвах своята кариера на разработчик на софтуер преди повече от 15 години, намирането на **цялостен източник**, от който да науча тези неща беше невъзможно. Да, тогава имаше книги за отделните програмни езици, но те описваха единствено техния синтаксис. За описание на приложно-програмния интерфейс трябваше да се ползва самата документация към библиотеките. Имаше отделни книги посветени единствено на обектно-ориентираното програмиране. Различни алгоритми и структури от данни пък се преподаваха в университета. За качествен програмен код не се говореше въобще.

Научаването на всички тези неща „на парче" и усилията по събирането им в единен контекст си оставаше работа на избралия „**пътя на програмиста**". Понякога един такъв самообразоващ се програмист не успява да запълни огромни пропуски в познанията си просто защото няма идея за тяхното съществуване. Нека ви дам един пример, за да илюстрирам проблема.

През 2000 г. поех един **голям Java проект**. Екипът, който го разработваше беше от 25 души и до момента по проекта имаше написани приблизително 4 000 Java класа. Като ръководител на екипа, част от моята работа включваше редовното преглеждане на кода написан от другите програмисти. Един ден видях как един от моите колеги беше решил стандартната задача по **сортиране на масив**. Той беше написал отделен метод от около 25 реда, който реализираше тривиалния алгоритъм за **сортиране по метода на мехурчето**. Когато отидох при него и го запитах защо е направил това вместо да реши проблема на един единствен ред използвайки `Arrays.sort()`, той се впусна в обяснения как вградения метод е потромав и е по-добре тези неща да си ги пишеш сам. Накарах го да отвори документацията и му показах, че „тромавият" метод работи със сложност O(n*log(n)), а неговото мехурче е еталон за лоша производителност със своята сложност O(n*n). В следващите няколко минути от нашия разговор направих и истинското откритие – **моят колега нямаше идея какво е сложност на алгоритъма**, а самите му познания по стандартни алгоритми бяха трагични. В последствие открих, че той е завършил съвсем друг тип инженерна специалност, а не информатика. В това, разбира се, няма абсолютно нищо лошо. В познанията си по Java той не отстъпваше на останалите колеги, които имаха по-дълъг практически опит от него. Но в този ден ние открихме **празнина в неговата квалификация на разработчик**, за която той не беше и подозирал.

Не искам да оставате с погрешни впечатления от тази история. Въпреки, че един студент издържал успешно основните си изпити по **специалност "Информатика"** със сигурност ще знае базовите алгоритми за сортиране и ще може да изчисли тяхната сложност, той също ще има своите пропуски. Тъжната истина е, че в България университетското образование по тази специалност все още е **с твърде теоретична насоченост**. То твърде малко се е променило за последните 15 години. Да, програмите вече се пишат на Java и C#, но това са същите програми, които се пишеха тогава на Pascal и Ada. Преди около месец приех за консултация студент първокурсник, който следваше в специалност „Информатика" на един от най-големите държавни университети в България. Когато седнахме да прегледаме заедно записките му от лекциите по „**Увод в програмирането**" бях изумен от примерния код даван от преподавателя. Имената на методите бяха смесица от английски и транслитериран български. Имаше метод `calculate` и метод `rezultat`. Променливите носеха описателните имена `a1`, `a2`, и `suma`. Да, в този подход няма нищо трагично, докато се използва за примери от десет реда. Но когато този студент заеме след години своето заслужено място в някой голям проект, той ще бъде навикан от проектния ръководител, който ще му обяснява за **код конвенция**, **именуване с префикси**, **логическа**

свързаност на отговорностите и **диапазон на активност**. Тогава те заедно ще открият неговата празнина в познанията по качествен код по същия начин, по който ние с моя колега открихме проблемните му познания в областта на **алгоритмите**.

Скъпи читателю, смело мога да заявя, че **в ръцете си държиш една наистина уникална книга**. Нейното съдържание е подбрано изключително внимателно. То е подредено и поднесено **с внимание към детайла**, на който са способни само хора с огромен практически опит и солидни научни познания като авторите на тази книга. Години наред те също са се учили „в движение", допълвайки и разширявайки своите познания. Работили са години по **огромни софтуерни проекти**, участвали са в научни конференции, преподавали са на стотици студенти. Те знаят какво е нужно да знае всеки един, който се стреми към кариера в областта на разработката на софтуер **и са го поднесли така, както никоя книга по увод в програмирането** не го е правила до момента.

Твоето пътуване през страниците ще те преведе през синтаксиса на **езика Java**. Ще видиш използването на голяма част от приложно-програмния му интерфейс. Ще научиш основите на **обектно-ориентираното програмиране** и ще боравиш свободно с термини като обекти, събития и изключения. Ще видиш най-често използваните **структури от данни** като масиви, дървета, хеш-таблици и графи. Ще се запознаеш с най-често използваните **алгоритми** за работа с тези структури и ще узнаеш за техните плюсове и минуси. Ще разбереш концепциите по **конструиране на качествен програмен код** и ще знаеш какво да изискваш от програмистите си, когато някой ден станеш ръководител на екип. В допълнение книгата ще те предизвика с много практически задачи, които ще ти помогнат да усвоиш по-добре и **по пътя на практиката** материала, който се разглежда в нея. А ако някоя от задачите те затрудни, винаги ще можеш да погледнеш **решението**, което авторите предоставят за всяка от тях.

Програмистите правят грешки – от това никой не е застрахован. По-добрите грешат от недоглеждане или преумора, по-лошите от **незнание**. Дали ще станеш добър или лош разработчик на софтуер зависи изцяло от теб и най-вече от това, доколко си готов пос**тоянно да инвестираш в своите познания** – било чрез курсове, чрез четене или чрез практическа работа. Със сигурност обаче мога да ти кажа едно – **колкото и време да инвестираш в тази книга, няма да сгрешиш**. Ако преди няколко години някой, желаещ да стане разработчик на софтуер, ме попиташе „От къде да започна?" нямаше как да му дам еднозначен отговор. Днес мога без притеснения да заявя - „**Започни от тази книга!**".

От все сърце ти желая успех в овладяването на **тайните на Java и разработката на софтуер**!

Николай Манчев е консултант и **софтуерен разработчик** с дългогодишен опит в Java Enterprise и Service Oriented Architecture (SOA). Работил е за BEA Systems и Oracle Corporation. Той е сертифициран разработчик по програмите на Sun, BEA и Oracle. Преподава софтуерни

технологии и води курсове по Мрежово програмиране, J2EE, Компресия на данни и Качествен програмен код в ПУ "Паисий Хилендарски" и СУ "Св. Климент Охридски". Водил е редица курсове за разработчици по Oracle технологии в централна и източна Европа (Унгария, Гърция, Словакия, Словения, Хърватска и други) и е участвал в международни проекти по внедряване на **J2EE базирани системи** за управление на сигурността. Негови разработки в областта на алгоритмите за компресия на данни са приети и представяни в САЩ от IEEE. Николай е почетен член на Българска асоциация на разработчиците на софтуер (БАРС) – www.devbg.org. Можете да се свържете с Николай по email: nick {at} manchev.org.

Отзив от Павлин Добрев – ProSyst Labs

Казват, че **първата книга по програмиране е важна**, защото определя стила и начина, по който се изгражда един професионален програмист. Тази книга е посветена на **фундаменталните знания в програмирането** – на началото непроменено повече от 30 години и поднесено по един вълнуващ начин. **Струва си да започнете с нея**!

Книгата "Въведение в програмирането с Java" е отлично учебно пособие, което ви дава възможност **по лесен и достъпен начин да овладеете основите на програмирането**. Това е петата книга, написана под ръководството на Светлин Наков, и също както останалите, е изключително **ориентирана към практиката**. Тя е изпълнена с многобройни примери и практически съвети за решаване на основни задачи от ежедневната работа на един програмист. Водещият автор **Светлин Наков е известен с преподавателските си умения** и с отличния си усет как да обясни по интересен и разбираем начин дори най-сложните теми от разработката на софтуер. Неговият дългогодишен опит като преподавател и автор на технически статии и учебни материали са му позволили да подреди и структурира темите от книгата, така че да се възприемат максимално лесно от начинаещите.

Въпреки големия брой автори, всеки от които с различен професионален и преподавателски опит, между отделните глави на книгата се забелязва **ясна логическа свързаност**. Тя е написана **разбираемо**, с подробни обяснения и с много, много примери, далеч от сухия академичен стил, присъщ за повечето университетски учебници.

Насочена към прохождащите в програмирането, книгата поднася внимателно, **стъпка по стъпка**, най-важното, което един програмист трябва да владее, за да практикува професията си – започвайки от променливи, цикли и масиви и достигайки до **фундаменталните структури от данни** и **алгоритми**. Книгата засяга и важни теми като рекурсивни алгоритми, дървета, графи и хеш-таблици. Това е една от малкото книги, която същевременно учи на **добър програмен стил** и качествен програмен код. Отделено е достатъчно внимание на принципите на обектно-ориентираното програмиране и обработката на изключения, без които съвременната разработка на софтуер е немислима.

Книгата "Въведение в програмирането с Java" учи на важните **принципи и концепции в програмирането**, на начина, по който програмистите разсъждават логически, за да решават проблемите, с които се сблъскват в ежедневната си работа. Ако трябваше заглавието на книгата да съответства още по-точно на съдържанието ѝ, тя трябваше да се казва "**Фундаментални основи на програмирането**".

Тази книга не съдържа всичко за програмирането и **няма да ви направи експерти** по разработка на софтуер. За да станете наистина добри програмисти, ви трябва много, много практика. **Започнете от задачите за упражнения след всяка глава**, но не се ограничавайте само с тях. Ще изпишете хиляди редове програмен код докато наистина станете добри – такъв е животът на програмиста. Тази книга е **наистина силен старт**! Възползвайте се от възможността да намерите всичко най-важно на куп, без да се лутате из хилядите самоучители и статии в Интернет. На добър път!

Д-р Павлин Добрев е технически директор на фирма Просист Лабс (www.prosyst.com), **софтуерен инженер с повече от 15 години опит**, консултант и учен, доктор по Компютърни системи, комплекси и мрежи. Павлин има световен принос в развитието на съвременните компютърни технологии и технологични стандарти. Той участва активно в международни стандартизационни организации като OSGi Alliance (www.osgi.org) и Java Community Process (www.jcp.org), както и инициативи за софтуер с отворен код като Eclipse Foundation (www.eclipse.org). Павлин управлява софтуерни проекти и консултира фирми като Miele, Philips, Siemens, BMW, Bosch, Cisco Systems, France Telecom, Renault, Telefonica, Telekom Austria, Toshiba, HP, Motorola, Ford, SAP и др. в областта на вградени приложения, OSGi базирани системи за автомобили, мобилни устройства и домашни мрежи, среди за разработка и **Java Enterprise** сървъри за приложения. Той има много научни и технически публикации и е участник в престижни международни конференции.

Отзив от Васил Поповски – VMWare

Със сигурност **бих искал да имам книга като тази, когато самият аз навлизах в програмирането**. Това без съмнение е една от най-добрите книги по програмиране за начинаещи и покрива фундаментални знания, които ще ползвате през цялата си кариера на софтуерни разработчици.

Като служител с ръководна роля във фирма VMware и преди това в Sciant често ми се налага да правя технически интервюта на кандидати за работа в нашата фирма.

Учудващо е колко голяма част от **кандидатите за софтуерни инженери не владеят фундаментални основи на програмирането**. Случва се кандидати с дългогодишен опит да не могат да нарисуват свързан списък, да не знаят как работи хеш-таблицата, да не са чували какво е сложност на алгоритъм, да не могат да сортират масив или да го сортират, но със сложност $O(n^3)$. Направо не е за вярване колко много **самоуки**

програмисти има, които не владеят фундаменталните основи на програмирането, които ще намерите в тази книга. Много от практикуващи професията софтуерен разработчик не са наясно дори с най-основните структури от данни в програмирането и не знаят как да обходят дърво с рекурсия. **За да не бъдете като тях, прочетете тази книга**! Тя е първото учебно пособие, от което трябва да започнете своето развитие като програмисти. **Фундаменталните познания** по структури от данни и алгоритми, които ще намерите в тази книга ще са ви необходими, за да изградите успешно кариерата си на софтуерен разработчик.

Ако започнете от правене на динамични уеб сайтове с бази от данни и AJAX, без да знаете какво е свързан списък, дърво или хеш-таблица, един ден ще разберете какви фундаментални пропуски в знанията си имате. **Трябва ли да се изложите на интервю за работа**, пред колегите си или пред началника си, когато се разбере, че не знаете за какво служи хеш-кодът или как работи `ArrayList` или как се обхождат рекурсивно директориите по твърдия диск?

Повечето книги за програмиране ще ви научат да пишете прости програмки, но няма да обърнат внимание на **качеството на програмния код**. Това е една тема, която повечето автори смятат за маловажна, но писането на качествен код е основно умение, което **отличава кадърните от посредствените програмисти**. С годините можете и сами да стигнете до добрите практики, които тази книга ще ви препоръча, но трябва ли да се учите по метода на пробите и грешките? Тази книга ще ви даде лесния начин да тръгнете в правилната посока – да овладеете базовите **структури от данни и алгоритми**, да се научите да мислите правилно и да пишете кода си качествено. Пожелавам ви ползотворно четене.

Васил Поповски е **софтуерен архитект** във VMware България (www.vmware.com) с повече от 10 години професионален опит като Java разработчик. Във VMware България се занимава с разработка на скалируеми, enterprise Java системи. Преди това е работил като старши мениджър във VMware България, като технически директор във фирма Sciant и като ръководител екип в SAP Labs България.

Като ученик Васил е **печелил призови отличия** в редица национални и международни състезания и е носител на бронзов медал от Международната олимпиада по информатика, Сетубал, 1998 и бронзов медал от Балканиада по информатика, Драма, 1997. Като студент Васил участвал в редица национални студентски състезания и в световното междууниверситетско състезание по програмиране (ACM ICPC). През 2001/2002 води курса "Обработване на транзакции" в СУ "Св. Климент Охридски". Васил е един от учредителите на Българска асоциация на разработчиците на софтуер (БАРС) – www.devbg.org.

Отзив от Веселин Райчев – Google

Ако искате с първата си книга по програмиране да добиете солидни знания, които ще са ви полезни години наред, то **това трябва да е вашата първа книга**!

Като инженер в Google съм провеждал **десетки технически интервюта** и трябва да ви споделя, че кандидати, които не се справят с материала от тази книга, се отхвърлят. Наблюдавал съм, че други големи компании също имат подобна практика. Причините са много прости – без да знаете прости **структури от данни**, не можете да участвате в дизайна на по-сложни проекти. Без основите на **обектно-ориентираното програмиране** няма да можете да работите в екип. Ако не знаете езика и стандартните библиотеки, няма да можете да работите бързо, но аз поставям това чак на трето място. Към всеки език има достатъчно документация, която най-често може да се чете докато се създава проекта. Наблюдавал съм и съм изпитвал на собствен гръб как с **добри основни познания по програмиране** човек научава нови императивни езици като **Python** или **JavaScript** за два дни. Когато имате добра основа, технологиите ще ги научите.

Вероятно и без да прочетете тази книга ще можете да работите като софтуерен разработчик, просто **ще ви е много по-трудно**. Наблюдавал съм случаи на преоткриване на колелото, много често в по-лош вид от теоретично най-доброто и най-често целият екип губи от това. Всеки, занимаващ се с програмиране, рано или късно **трябва да прочете какво е сложност на алгоритъм, какво е хеш-таблица и какво е двоично търсене**. Защо не започнете още отсега като прочетете тази книга?

Съществуват много книги за Java и още повече за програмиране. За много от тях ще кажат, че са най-доброто ръководство, най-бързо навлизане в езика. Тази книга е различна с това, че ще ви покаже **какво трябва да знаете, за да постигате успехи**, а не какви са тънкостите на езика Java. Ако смятате темите в тази книга за безинтересни, вероятно софтуерното инженерство просто не е за вас.

Веселин Райчев е софтуерен инженер в Google, където се занимава с Google Maps и Google Translate. Преди това е работил в Motorola Biometrics и Metalife AG.

Веселин е печелил **призови отличия** в редица национални и международни състезания и е носител на бронзов медал от Международната олимпиада по информатика, Южна Корея, 2002 и сребърен медал от Балканиада по информатика. Два пъти е представял СУ "Св. Климент Охридски" на **световни финали по информатика** (ACM ICPC) и е преподавал в няколко изборни курса във Факултета по математика и информатика на СУ.

Отзив от Димитър Костов – SAP Labs Bulgaria

Най-хубавото на тази книга е, че използва Java само като пример и илюстрация на най-важното – **базовите знания, които отличават Програмиста от обикновения Кодер**, а именно основите на програмирането – структури от данни и алгоритми. С тази книга **можете да се научите наистина да програмирате**, а между другото и да научите езика Java.

Често на интервюта за работа идват хора, които могат да те зашеметят с познания по Java, C#, Flash, SQL и т. н. – безброй технологии, нови, стари, за какво ли не. Необходимо, но не достатъчно условие за добра кариера. По две причини: **само след година днешните технологии ще са стари** и само след година новото поколение кандидати ще познава точно толкова добре *новите* технологии. Какво помага на успешните софтуерни инженери да издържат на тази динамика и конкуренция? Какво ги прави наистина ценни и защо са успешни? Защото те знаят и разбират основите на софтуера, а той се изгражда около **структури от данни и алгоритми**. Това им позволява да усвояват нови концепции и технологии за много кратко време и да ги прилагат ефективно – било то обектно-ориентирани езици като Java или C# или технологии като J2EE и AJAX и с тяхна помощ да решат проблеми, да създадат завършен и готов за ползване, **професионално направен софтуер**.

Хубавото на тази книга е, че започва от **принципите на програмирането**, позволява ви да добиете здрава основа, от която да можете да се развивате в произволна посока – уеб, middleware, бизнес софтуер, embedded, игри, бази данни – каквото ви е интересно и ви влече. Ще се убедите, че „**езикът няма значение**", а ще научите и Java.

Тази книга е **за всеки, който иска сериозно да се занимава с правене на софтуер**. Извлечете максимална полза от нея. Не подминавайте с лека ръка разделите, които на пръв поглед ви се виждат ненужни или неприложими. Да, конзолните приложения не са върха на модата в момента, но покрай тях ще научите множество полезни алгоритми и концепции, които с изненада можете да срещнете при писането на сървърни приложения например.

Ако в някакъв момент материалът ви се стори твърде сух и ви доскучае, прескочете бързо до някоя от **главите „за професионалисти"**, 19, 20, 21 или 22 да почерпите вдъхновение и смисъл, а после се върнете там, до където сте стигнали. Така може няколко пъти да прочетете тези глави (от 19 до 22), което ще ви е само от голяма полза.

... и **правете упражненията**! Така ще затвърдите знанията си и ще натрупате опит с Eclipse и Java, така че, когато започнете да работите по реални проекти рядко да ви се налага да ровите в документацията.

Димитър Костов е **мениджър на отдел** в SAP Labs Bulgaria, където заедно с екипите си се занимава със създаването и реализацията на средата за администриране и управление на SAP NetWeaver Java application server. Преди това е бил в топ мениджмънт екипа на аутсорсинг компании,

където е започнал като програмист. Работил е като ИТ консултант в различни сфери – строителство, производство, финанси, ИТ и други. Ръководил е различни по големина екипи и е разработвал софтуер за множество фирми като Mercury Interactive, Jaguar Formula 1, Ford WRC, DHL, Belair и други. Завършил е НПМГ и Софийски Университет, ФМИ, специалност Информатика. Занимава се професионално със софтуер от 1992 г.

Отзив от Явор Ташев – Microsoft

Книгата "Въведение в програмирането с Java" съдържа **фундаменталните първоначални познания, от които всеки начинаещ програмист има нужда**, представени в компактен вид, даващ възможност за лесно и бързо усвояване.

Никои не се е родил научен, всеки е започнал от начало. Тази книга е едно добро начало за всеки, който има желанието, търпението и упоритостта да започне да се занимава с програмиране. Създаването на **правилната основа от знания е най-важната предпоставка за по-нататъшното професионално развитие**. Когато я има основата, научаването на конкретен програмен език или софтуерна технология е съвсем просто, тъй като основните съвременни езици за програмиране много си приличат, а технологиите следват едни и същи принципи, които всеки програмист постепенно научава в практиката си. В основата на програмирането стоят **структурите от данни, алгоритмите и правилното логическо инженерно мислене**.

Настоящата книга е хубаво въведение, което не се фокусира върху детайлите на конкретен език за програмиране, а върху **концепциите и принципите на инженерното мислене**, структурите от данни, качествен програмен код и решаването на проблеми от практиката. Тази книга ще ви покаже как не просто да набивате програмен код, а да програмирате правилно и **да мислите, като истински професионалисти**.

В никакъв случай не очаквайте, че прочитането на тази книга ви прави програмист! Това е само **началото на дългото пътуване** в света на разработката на софтуер. Натрупването на познания и умения е един непрекъснат и най-хубавото – безкраен процес. Добре дошъл на борда читателю!!!

Явор Ташев, софтуерен инженер, Microsoft

Отзив от Любомир Иванов – Mobiltel

Началото винаги е трудно, а може да бъде и още по-трудно – достатъчно е да не знаеш от къде да почнеш или да си създадеш грешен подход. **Тази книга е за всички, които сега започват и искат да бъдат добри програмисти**, а също и за всички самоусъвършенстващи се програмисти, желаещи да запълнят пропуските, които със сигурност имат.

Ако преди 5 или 10 години някой ми беше казал, че съществува книга, от която да научим основите на управлението на хора и проекти –

бюджетиране, финанси, психология, планиране и т.н., нямаше да му повярвам. Не бих повярвал и днес. За всяка от тези теми има десетки книги, които трябва да бъдат прочетени.

Ако преди един месец някой ми беше казал, че съществува книга, от която можем да научим основите на програмирането, необходими на всеки софтуерен разработчик – пак нямаше да му повярвам.

Спомням си времето като начинаещ програмист и студент – четях **няколко книги за езици за програмиране**, други за алгоритми и структури от данни, а трети за писане на качествен код. Много малко от тях ми помогнаха да мисля алгоритмично и да си изградя подход за решаване на ежедневните проблеми, с които се сблъсквах в практиката. Нито една не ми даде цялостен поглед над всичко, което исках и трябваше да знам като програмист и софтуерен инженер. Единственото, което помагаше беше инатът и **преоткриването на колелото**.

Днес чета тази книга и се радвам, че най-сетне, макар и малко късно за мен, **някой се е хванал и е написал Книгата**, която ще помогне на всеки начинаещ програмист да сглоби **големия пъзел на програмирането** – модерен език за програмиране, структури от данни, качествен код, алгоритмично мислене и решаване на проблеми. Това е **книгата, от която трябва да за почнете с програмирането**, ако искате да овладеете изкуството на качественото програмиране.

Тази книга не е само за начинаещите. Дори програмисти с няколко годишен опит има какво да научат от нея. **Препоръчвам на всеки разработчик на софтуер**, който би искал да разбере какво не е знаел досега.

Приятно четене!

Любомир Иванов е ръководител на отдел "Data Services Research and Development" в Мобилтел ЕАД, където се занимава с разработка и внедряване на ИТ решения за telecom индустрията.

Отзив от Стамен Кочков – SAP Labs Bulgaria

Тази книга е много повече от въведение! С впечатляващото като обем и правилно поднесено от авторите съдържание, тя наистина **дава изчерпателни познания по основите на програмирането**. Удачният избор на платформата я прави още по-достъпна за начинаещите и дава в ръцете на прочелите я инструмент, с който не само да започнат, но и да продължат развитието си в ИТ индустрията.

Стамен Кочков е директор "Програмиране" във фирма SAP Labs Bulgaria, която е подразделение на софтуерния гигант SAP (www.sap.com). Стамен има над **15 години опит в ИТ индустрията** като софтуерен разработчик, мениджър на проекти и ръководител отдел. Стамен следи Java базираните технологии още от зараждането им и активно е участвал в разработката както и управлението на успешни ИТ проекти базирани на нея.

Отзив от Станислав Овчаров – Musala Soft

Мащабно и увлекателно **въведение в професионалното програмиране**! Книгата се разпростира от базови и класически програмистки концепции, през индустриални технологии и похвати, до екзотични, но уникални теми като **наръчник за решаване на задачи по програмиране**. Това не е поредното сухо изложение! Четенето на тази книга остава усещане за диалог с авторите. Темите са представени в достъпен и свободен стил, със силен отпечатък от личното мнение на авторите и техния професионален опит. Едно **свежо и полезно попълнение за личната Ви библиотека**!

Станислав Овчаров, Технически Директор, Мусала Софт

Отзив от Христо Дешев – telerik

Учудващо е, че голям процент от програмистите не обръщат внимание на **малките н**еща като имената на променливите и **добрата структура на кода**. Тези неща се натрупват и накрая формират разликата между добре написания софтуер и купчината спагети. **Тази книга учи на дисциплина** и "хигиена" в писането на код още с основите на програмирането, а това несъмнено ще Ви изгради като професионалист.

Христо Дешев, софтуерен инженер, Телерик АД

Отзив от Драгомир Николов – Software AG

Една прекрасна книга, която **изчерпателно запознава читателя, с основите на програмирането**, използвайки най-популярния език Java. Задължително четиво за всеки търсещ успешна професионална реализация като програмист.

Драгомир Николов, мениджър разработка,
webMethods RnD, Software AG

Отзив от Панайот Добриков – SAP AG

Настоящата книга е едно **изключително добро въведение в програмирането** за начинаещи и водещ пример в течението (промоцирано от Wikipedia и други) да се създава и разпространява достъпно за всеки знание не само **безплатно**, но и с изключително високо качество.

Панайот Добриков, Програмен директор в SAP AG
Автор на книгата "Програмиране=++Алгоритми;"

Книгата е безплатна!

Настоящата книга се разпространява **напълно безплатно** в електронен вид по лиценз, който позволява използването ѝ за всякакви цели, включително и в комерсиални проекти. Книгата се разпространява и в хартиен вид срещу заплащане, което покрива разходите по отпечатването и разпространението ѝ, без да се реализира печалба.

Спонсори

Авторският колектив благодари на **спонсорите**, които подпомогнаха първото издание на книгата на хартия:

telerik
deliver more than expected

SAP Labs Bulgaria

ТОВА Е ТВОЯТ ГЛАС. Mtel

Спонсор на поредното издание на Intro Java книгата на хартия (от 2017 г.) е Софтуерният университет (СофтУни):

Сайтът на книгата

Официалният уеб сайт на книгата "Въведение в програмирането с Java" е достъпен от адрес: http://www.introprogramming.info. От него можете да изтеглите цялата книга в електронен вид, сорс кода на примерите и други ресурси.

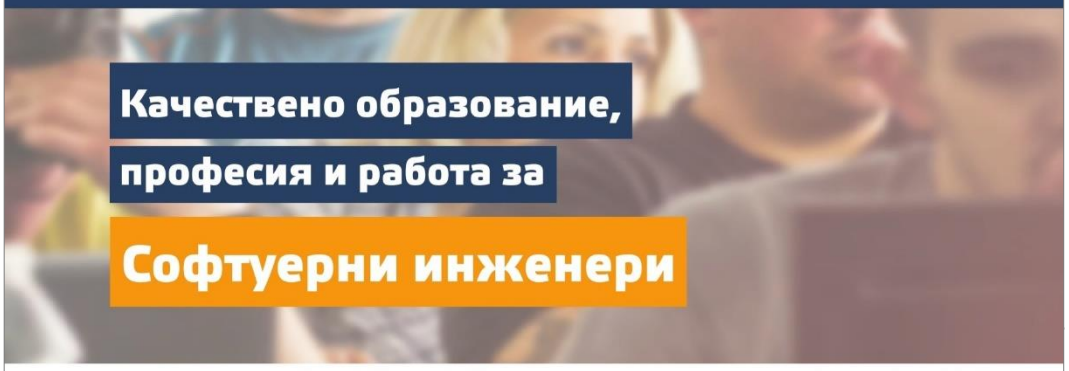
Допълнителни ресурси

Към настоящата книга са разработени **допълнителни ресурси**, които се разпространяват под свободен лиценз:

- Презентационни **слайдове и видео** по основи на програмирането с Java и други езици: https://softuni.bg/courses/programming-basics.
- Презентационни **слайдове, упражнения и видео** по Java за напреднали: https://softuni.bg/courses/java-advanced.
- **Слайдове и видео** по Java обектно-ориентирано програмиране: https://softuni.bg/courses/java-oop-basics (основи на ООП с Java), https://softuni.bg/courses/java-oop-advanced (ООП за напреднали).
- **Форум** за начинаещи в програмирането, където можете да зададете вашите въпроси по учебния материал от книгата и да получите адекватни отговори бързо. Форумът се задвижва от студентите в Софтуерния университет: https://softuni.bg/forum.

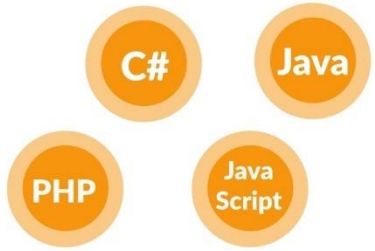
Светлин Наков,
София, 14.07.2008 г.
(последна редакция: 7.04.2017 г.)

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 1. Въведение в програмирането

Автор

Лъчезар Цеков

В тази тема...

В настоящата тема ще разгледаме основните термини от програмирането и ще напишем първата си програма. Ще се запознаем с това какво е програмиране и каква е връзката му с компютрите и програмните езици.

Накратко ще разгледаме основните етапи при писането на софтуер.

Ще въведем езика Java и ще се запознаем с Java платформата и Java технологиите.

Ще разгледаме какви помощни средства са ни необходими, за да можем да програмираме на Java.

Ще използваме Java, за да напишем първата си програма, ще я компилираме и изпълним както от командния ред, така и от среда за разработка Eclipse.

Ще се запознаем с документацията на Java API, която позволява по-нататъшно изследване на възможностите на езика.

Какво означава "да програмираме"?

В днешно време компютрите навлизат все по-широко в ежедневието ни и все повече имаме нужда от тях, за да се справяме със сложните задачи на работното място, да се ориентираме, докато пътуваме, да се забавляваме или да общуваме. Неизброимо е приложението им в бизнеса, в развлекателната индустрия, в далекосъобщенията и в областта на финансите. Няма да преувеличим, ако кажем, че компютрите изграждат нервната система на съвременното общество и е трудно да си представим съществуването му без тях.

Въпреки масовото им използване, малко хора имат представа как всъщност работят компютрите. Всъщност не компютрите, а програмите, които вървят на тях (софтуерът), имат значение. Програмите са тези, които са полезни за потребителите, и чрез тях се реализират различните типове услуги, променящи живота ни.

Структура на компютъра

За да разберем какво значи да програмираме, нека грубо да сравним компютъра и операционната система, вървяща на него, с едно голямо предприятие заедно с неговите цехове, складове и транспортни механизми. Това сравнение е грубо, но дава възможност да си представим степента на сложност на един съвременен компютър. В компютъра работят много процеси, които съответстват на цеховете и поточните линии. Твърдият диск заедно с файловете на него и оперативната (RAM) памет съответстват на складовете, а различните протоколи са транспортните системи, внасящи и изнасящи информация.

Различните видове продукция в едно предприятие се произвеждат в различните цехове. Цеховете използват суровини, които взимат от складовете, и складират готовата продукция обратно в тях. Суровините се транспортират в складовете от доставчиците, а готовата продукция се транспортира от складовете към пласмента. За целта се използват различни видове транспорт. Материалите постъпват в предприятието, минават през различни стадии на обработка и напускат предприятието, преобразувани под формата на продукти. Всяко предприятие преобразува суровините в готов за употреба продукт.

Компютърът е машина за обработка на информация и при него както суровината, така и продукцията е информация. Входната информация най-често се взима от някой от складовете (файлове или RAM памет), където е била транспортирана, преминава през обработка от един или повече процеси и излиза модифицирана като нов продукт. Типичен пример за това са уеб базираните приложенията. При тях за транспорт както на суровините, така и на продукцията се използва протоколът HTTP.

Управление на компютъра

Целият процес на изработка на продуктите има много степени на управление. Отделните машини и поточни линии се управляват от оператори, цеховете се управляват от управители, а предприятието като цяло се управлява от директори. Всеки от тях упражнява контрол на различно ниво. Най-ниското ниво е това на машинните оператори – те управляват машините, образно казано, с помощта на копчета и ръчки. Следващото ниво е на управителите на цехове. На най-високо ниво са директорите, те управляват различните аспекти на предприятието. Всеки от тях управлява, като издава заповеди.

По аналогия при компютрите и софтуера има много нива на управление. На най-ниско машинно ниво се управлява самият процесор и регистрите на компютъра – можем да сравним това с управлението на машините в цеховете. На по-високо системно ниво се управляват различните аспекти на операционната система като файлова система, периферни устройства, протоколи – можем да сравним това с управлението на цеховете и отделите в предприятието. На най-високо ниво в софтуера са приложенията. При тях се управлява цял ансамбъл от процеси, за изпълнението на които са

необходими огромен брой операции на процесора. Това е нивото на директорите, които управляват цялото предприятие с цел максимално ефективно използване на ресурсите за получаване на качествени продукти.

Същност на програмирането

Същността на програмирането е да се управлява работата на компютъра на всичките му нива. Управлението става с помощта на заповеди (команди) от програмиста към компютъра. Да програмираме, означава да управляваме компютъра с помощта на заповеди. Заповедите се издават в писмен вид и биват безпрекословно изпълнявани от компютъра. Те могат да бъдат подписани и подпечатани с цел да се удостовери авторитета на този, който ги издава.

Програмистите са хората, които издават заповедите. Заповедите са много на брой и за издаването им се използват различни видове програмни езици. Всеки език е ориентиран към някое ниво на управление на компютъра. Има езици, ориентирани към машинното ниво – например асемблер, други са ориентирани към системното ниво, например C. Съществуват и езици от високо ниво, ориентирани към писането на приложни програми. Такива са Java, C++, C#, Visual Basic, Python, Ruby, PHP и други.

В тази книга ще разгледаме програмния език Java. Това е език за програмиране от високо ниво. При използването му позицията на програмиста в компютърното предприятие се явява тази на директора.

Етапи при разработката на софтуер

Писането на софтуер може да бъде сложна задача, която отнема много време на цял екип от софтуерни инженери и други специалисти. Затова с времето са се обособили методики и практики, които улесняват живота на програмистите. Разработката на всеки софтуерен продукт преминава през няколко етапа, а именно:

- Събиране на изискванията за продукта и изготвяне на задание;
- Планиране и изготвяне на архитектура и дизайн;
- Реализация;
- Изпитания на продукта (тестове);
- Внедряване и експлоатация;
- Поддръжка.

Фазите реализация, изпитания, внедряване и поддръжка се осъществяват с помощта на програмиране.

Събиране на изискванията и изготвяне на задание

В началото съществува само идеята за определен продукт. Това е набор от изисквания, дефиниращи действия от страна на компютъра, които в общия случай улесняват досега съществуващи дейности. Като пример може да

дадем изчисляването на заплатите, пресмятане на балистични криви, търсене на най-пряк път в Google Maps. Много често софтуерът реализира несъществуваща досега функционалност като автоматизиране на някаква дейност и др.

Изискванията за продукта най-често се дефинират под формата на документи, написани на естествен език – български, английски или друг. На този етап не се програмира, дори не са необходими програмисти. Изискванията се дефинират от експерти, запознати с проблематиката на конкретната област, които умеят да ги описват в разбираем за програмистите вид. В общия случай такива експертите не разбират от програмиране. Те се наричан **бизнес анализатори**.

Планиране и изготвяне на архитектура и дизайн

След като изискванията бъдат събрани, идва ред на етапа по анализ на изискванията и планиране. Този етап включва съставяне на технически план за изпълнението на проекта, който описва платформите, технологиите и първоначалната архитектура (дизайн) на програмата. Тази стъпка включва значителна творческа работа и обикновено се реализира от софтуерни инженери с голям опит, наричани понякога **софтуерни архитекти**. Съобразно изискванията се избират:

- Вида на приложението – например конзолно приложение, настолно приложение (GUI application), клиент-сървър приложение, уеб приложение, Rich Internet Application (RIA) или peer-to-peer приложение;
- Архитектурата на програмата – например еднослойна, двуслойна, трислойна, многослойна или SOA архитектура;
- Програмният език, най-подходящ за реализирането – например Java или C++ или C# или комбинация от езици;
- Технологиите, които ще се ползват: платформа (примерно Java EE, Microsoft .NET или друга), сървъри за бази данни (примерно Oracle, SQL Server, MySQL или друга), технологии за потребителски интерфейс (примерно Flash, JavaServer Faces, Eclipse RCP, ASP.NET, Windows Forms или друга), технологии за достъп до данни (примерно Hibernate, JPA или LINQ to SQL), технологии за изготвяне на отчети (примерно Jasper Reports) и много други технологии, които ще бъдат използвани за реализирането на различни части от приложението.
- Броят и уменията на хората, които ще съставят екипа за разработка (големите и сериозни проекти се пишат от големи и сериозни екипи от разработчици);
- План на разработката – етапи, на които се разделя функционалността, и срокове за всеки етап.
- Други (големина на екипа, местоположение на екипа и т.н.).

Въпреки че съществуват много правила, спомагащи за правилния анализ и планиране, на този етап се изискват значителна интуиция и усет. Тази стъпка предопределя цялостното по-нататъшно развитие на процеса на разработка. На този етап не се извършва програмиране, а само подготовка за него.

Реализация

Етапът, най-тясно свързан с програмирането, е етапът на реализацията (имплементацията). Съобразно със заданието, дизайна и архитектурата на програмата (приложението) се пристъпва към реализирането (написването) ѝ в програмен вид. Този етап се изпълнява от програмисти, които пишат програмния код.

Изпитания на продукта (тестове)

Важен етап от разработката на софтуер е етапът на изпитания на продукта. Той цели да удостовери, че реализацията следва и покрива изискванията на заданието. Този процес може да се реализира ръчно, но предпочитаният вариант е написването на автоматизирани тестове, които да реализират проверките. Тестовете са малки програми, които автоматизират, до колкото е възможно, изпитанията. Съществуват парчета функционалност, за които е много трудно да се напишат тестове и поради това процесът на изпитание на продукта включва както автоматизирани, така и ръчни процедури.

Процесът на тестване (изпитание) се реализира от екип инженери по осигуряването на качеството – quality assurance (QA) инженери. Те работят в тяхно взаимодействие с програмистите за откриване и коригиране на дефектите (бъговете). В този етап почти не се пише нов програмен код, а само се оправят дефекти в съществуващия код.

В процеса на изпитанията най-често се откриват множество пропуски и грешки и програмата се връща обратно в етап на реализация. До голяма степен етапите на реализация и изпитания вървят ръка за ръка и е възможно да има множество преминавания между двете фази, преди продуктът да е покрил изискванията на заданието и да е готов за етапа на внедряване и експлоатация.

Внедряване и експлоатация

Внедряването или инсталирането е процесът на въвеждане на даден софтуерен продукт в експлоатация. Ако продуктът е сложен и обслужва много хора, този процес може да се окаже най-бавният и най-скъпият. За по-малки програми това е относително бърз и безболезнен процес. Най-често се разработва специална програма – инсталатор, която спомага за по-бързата и лесна инсталация на продукта. Понякога, ако продуктът се внедрява в големи корпорации с десетки хиляди копия, се разработва допълнителен поддържащ софтуер специално заради внедряването. Като пример можем да дадем внедряването на Microsoft Windows в българската

държавна администрация. След като внедряването приключи, продуктът е готов за експлоатация.

Внедряването се извършва обикновено от екипа, който е разработил продукта или от специално обучени специалисти по внедряването. В този етап почти не се пише нов код, но съществуващият код може да се доработва и конфигурира докато покрие специфичните изисквания за успешно внедряване.

Поддръжка

В процеса на експлоатация неминуемо се появяват проблеми – заради грешки в самия софтуер или заради неправилното му използване и конфигурация или заради промени в нуждите на потребителите. Тези проблеми довеждат до невъзможност за употреба на продукта и налагат допълнителна намеса от страна на разработчиците и експертите по поддръжката. Процесът по поддръжка обикновено продължава през целия период на експлоатация независимо колко добър е софтуерният продукт.

Поддръжката се извършва от екипа по разработката на софтуера. В зависимост от промените, които се правят, в този процес могат да участват бизнес анализатори, архитекти, програмисти, QA инженери, администратори и други.

Документация

Етапът на документацията всъщност не е отделен етап, а съпътства всички останали етапи. Документацията е много важна част от разработката на софтуер и цели предаване на знания между различните участници в разработката и поддръжката на продукта. Информацията се предава както между отделните етапи, така и в рамките на един етап.

Разработката на софтуер не е само програмиране

Както сами се убедихте, разработването на софтуер не е само програмиране и включва много други процеси като анализ на изискванията, проектиране, планиране, тестване и поддръжка, в които участват не само програмисти, но и много други специалисти. Програмирането е само една малка, макар и много съществена, част от разработката на софтуера.

В настоящата книга ще се фокусираме само и единствено върху програмирането, което е единственото действие от изброените по-горе, без което не можем да разработваме софтуер.

Нашата първа Java програма

Преди да преминем към подробно описание на езика Java и на Java платформата, нека да се запознаем с прост пример на това какво представлява една програма, написана на Java.

```
class HelloJava {
   public static void main(String[] arguments) {
      System.out.println("Hello, Java");
   }
}
```

Единственото нещо, което прави тази програма, е да изпише съобщението "Hello, Java" в стандартния изход. Засега е още рано да я изпълняваме, а само искаме да разгледаме структурата. Малко по-нататък ще дадем пълно описание на това как да се компилира и изпълни както от командния ред, така и от среда за разработка.

Как работи нашата първа Java програма?

Нашата първа програма е съставена от три логически части:

- Дефиниция на клас;
- Дефиниция на метод **main()**;
- Съдържание на метода **main()**.

Дефиниция на клас

На първия ред от нашата програма дефинираме клас с името **HelloJava**. Най-простата дефиниция на клас се състои от ключовата дума **class,** следвана от името на класа. В нашия случай името на класа е **HelloJava.**

Дефиниция на метод main()

На втория ред дефинираме функция (**метод**) с името **main()**, която представлява входна или стартова точка за програмата. Всяка програма на Java стартира от метод **main()** със сигнатура:

```
public static void main(String[] arguments)
```

Методът трябва да е деклариран по точно показания начин, трябва да е **public**, **static** и **void**, трябва да има име **main** и като списък от параметри трябва да има един единствен параметър от тип масив от **String**. Местата на модификаторите **public** и **static** могат да се разменят. В нашия пример параметърът се казва **arguments**, но това не е задължително, параметърът може да има произволно име. Повечето програмисти избират за име **args** или **argv**.

Ако някое от гореспоменатите изисквания не е спазено, програмата ще се компилира, но няма да може да се стартира, а ще ни даде съобщение за грешка, защото не съдържа стартова точка.

Съдържание на main() метода

Съдържанието на всеки метод се намира след сигнатурата на метода, заградено от отваряща и затваряща главни скоби. На третия ред от програмата ни използваме системния обект **System.out** и неговия метод **println(),** за да изпишем произволно съобщение в стандартния изход в случая "Hello, Java". Целият трети ред представлява един Java израз.

В **main()** метода можем да сложим произволна последователност от изрази и те ще бъдат изпълнени в реда, в който сме ги задали.

Подробна информация за изразите е дадена в главата "Оператори и изрази", работата с конзолата е описана в главата "Вход и изход от конзолата", а класовете и методите са описани в главата "Дефиниране на класове".

Java различава главни от малки букви!

В горния пример използвахме някои ключови думи, като **class**, **public**, **static** и **void** и имената на някои от системните обекти, като **System.out**.

Внимавайте, докато пишете! Изписването на един и същ текст с главни, малки букви или смесено в Java означава различни неща. Да напишем `Class` е различно от `class` и да напишем `System.out` е различно от `SYSTEM.OUT`.

Това правило важи за всички конструкции в кода – ключови думи, имена на променливи, имена на класове, стрингове и т.н.

Програмният код трябва да е правилно форматиран

Форматирането представлява добавяне на символи, несъществени за компилатора, като интервали, табулации и нови редове, които структурират логически програмата и улесняват четенето. Нека отново разгледаме кода на нашата първа програма.

```
class HelloJava {
   public static void main(String[] arguments) {
      System.out.println("Hello, Java");
   }
}
```

Програмата съдържа пет реда и някои от редовете са повече или по-малко отместени навътре с помощта на табулации. Всичко това можеше да се напише и без отместване:

```
class HelloJava {
public static void main(String[] arguments) {
System.out.println("Hello, Java");
```

```
}
}
```

или на един ред:

```
class HelloJava{public static void
main(String[]arguments){ System.out.println("Hello, Java");}}
```

или дори така:

```
   class
HelloJava
     {public
static void main(
        String[
] arguments) {
     System.
out.println("Hello, Java");}
               }
```

Горните примери ще се компилират и изпълнят по абсолютно същия начин като форматирания, но са далеч по-нечетливи и неудобни за промяна.

Не допускайте програмите ви да съдържат неформатиран код! Това силно намалява четимостта и довежда до трудно модифициране на кода.

Основни правила на форматирането

- Методите се отместват по-навътре от дефиницията на класа;
- Съдържанието на методите се отмества по-навътре от дефиницията на метода;
- Отварящата фигурна скоба **{** трябва да е на същия ред, на който е дефиниран класът или методът;
- Затварящата фигурна скоба **}** трябва да е сама на ред, отместена на същото разстояние като началото на реда на отварящата;
- Имената на класовете трябва да започват с главна буква;
- Имената на променливите и имената на методите трябва да започват с малка буква.

Имената на файловете съответстват на класовете

Всяка Java програма се дефинира в един или повече класа. Всеки публичен клас трябва да се дефинира в отделен файл с име, съвпадащо с името на класа и разширение **.java**. При неизпълнение на тези изисквания и опит за компилация получаваме съобщение за грешка.

Ако искаме да компилираме нашата първа програма, горния пример трябва да запишем във файл с името **`HelloJava.java`**.

Езикът и платформата Java

Първата версия на Java е разработена от Sun Microsystems и е пусната в употреба през 1995 година, като част от Java платформата. В последствие се появяват множество други реализации включително от GNU, Microsoft, IBM, Oracle и други технологични доставчици.

Независимост от средата

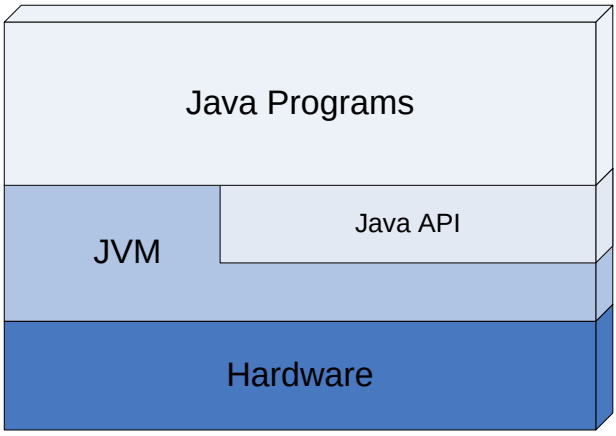
Основно предимство и причина, поради които езикът и платформата Java са се наложили, е възможността кодът веднъж компилиран да работи на произволни, поддържани от платформата, операционна система или хардуерно устройство. Можем да компилираме програмата на Windows и да я изпълняваме както на Windows така и на Linux, Apple OSX и всяка друга операционна система, поддържана от Java платформата. Можем дори да изпълняваме програмите на мобилните си телефони, поддържащи Java.

Bytecode

Идеята за независимост от средата е заложена при самото създаване на Java платформата и се реализира с малка хитрина. Изходният код не се компилира до инструкции, предназначени за даден микропроцесор, и не използва специфични възможности на дадена операционна система, а се компилира до междинен език - така нареченият **`bytecode`**. Този **`bytecode`** не се пуска за директно изпълнение от микропроцесора, а се изпълнява от негов аналог – виртуален процесор, наречен Java Virtual Machine (JVM).

Java Virtual Machine (JVM) – сърцето на Java

В самия център на Java платформата бие нейното сърце – Java Virtual Machine, която е основният компонент, осигуряващ независимостта от хардуер и операционна система.



JVM е абстрактна изчислителна машина. По аналогия на реалните електронноизчислителни машини има набор от инструкции и манипулира области от паметта по време на изпълнение на програмите.

Първият прототип на виртуалната машина е реализиран от Sun Microsystems за преносимо устройство, аналогично на съвременните персонални електронни помощници (PDA). Настоящата виртуална машина на Sun, компонент от продуктите Java 2 SDK и Java 2 Runtime Environment, емулира виртуална машина по далеч по-изтънчен начин на Win32 и Solaris платформи. Това не означава, че виртуалната машина по някакъв начин има зависимост от технологията за реализация, хардуер или операционна система. Виртуалната машина е реализирана програмно като интерпретатор на bytecode, но може да се реализира да компилира до инструкции на истинския процесор и дори да се изгради хардуерно базирана реализация – силициев микропроцесор, способен да изпълнява bytecode.

Виртуалната машина не знае за програмния езика Java, а само изпълнява инструкциите на bytecode, записани като class файлове. Всеки език за програмиране, който може да се компилира до bytecode, може да бъде изпълняван от виртуалната машина.

Езикът Java

Java е обектно-ориентиран език за програмиране от високо ниво с общо предназначение. Синтаксисът му е подобен на C и C++, но не поддържа много от неговите възможности с цел опростяване на езика, улесняване на програмирането и повишаване на сигурността. Програмите на Java представляват един или няколко файла с разширение **`.java`**. Тези файлове се компилират от компилатора **`javac`** до изпълним код и се записват във файлове със същото име, но различно разширение **`.class`**. Клас файловете съдържат Java bytecode инструкции, изпълним от виртуалната машина.

Ключови думи

Езикът Java използва следните ключови думи:

`abstract`	**`continue`**	**`for`**	**`new`**	**`switch`**
`assert`	**`default`**	**`goto`**	**`package`**	**`synchronized`**
`boolean`	**`do`**	**`if`**	**`private`**	**`this`**
`break`	**`double`**	**`implements`**	**`protected`**	**`throw`**
`byte`	**`else`**	**`import`**	**`public`**	**`throws`**
`case`	**`enum`**	**`instanceof`**	**`return`**	**`transient`**
`catch`	**`extends`**	**`int`**	**`short`**	**`try`**
`char`	**`final`**	**`interface`**	**`static`**	**`void`**
`class`	**`finally`**	**`long`**	**`strictfp`**	**`volatile`**

`const`	`float`	`native`	`super`	`while`

От тях две не се използват. Това са **`const`** и **`goto`**. Те са резервирани, в случай че се реши да влязат в употреба. Не всички ключови думи се използват още от създаването на първата версия на езика. Някои от тях са добавени в по-късните версии. Версия 1.2 добавя ключовата дума **`strictfp`**, версия 1.4 добавя ключовата дума **`assert`**, и версия 1.5 добавя ключовата дума **`enum`**.

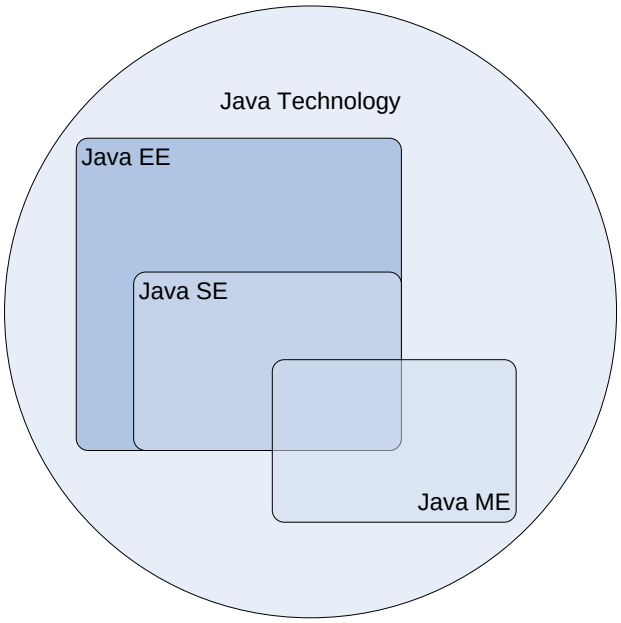
Основни конструкции в Java са класовете, методите, операторите, изразите, условните конструкции, цикли, типовете данни и изключенията.

Всичко това, както и повечето ключови думи, предстои да бъде разгледано подробно в следващите глави.

Автоматично управление на паметта

Едно от най-големите предимства на Java е предлаганото от нея автоматично управление на паметта. То предпазва програмистите от сложната задача сами да заделят памет за обектите и да следят подходящия момент за нейното освобождаване. Това рязко засилва производителността на програмистите и увеличава качеството на програмите, писани на Java.
За управлението на паметта се грижи специален компонент от виртуалната машина, наречен галено "**събирач на боклука**" или "**система за почистване на паметта**" (**Garbage Collector**). Основните задачи на събирача на боклука са да следи кога заделената памет за променливи и обекти вече не се използва, да освобождава тази памет и да я прави достъпна за последващи заделяния.

Java платформата



Java платформата, освен езика Java, съдържа в себе си Java виртуалната машина и множество помощни инструменти и библиотеки. Съществуват няколко разновидности на Java платформата съобразно целевата потребителска група, като възможностите, които те предлагат на разработчиците, се застъпват. На фигурата по-горе са изобразени версиите на Java за десктоп (SE), за мобилни устройства (ME) и за сървърни системи (EE).

Java Platform Standard Edition (Java SE)

Стандартната версия на Java платформата е предназначена за разработката и използването на настолни приложения. Тази версия включва в себе си:

- Java компилатор – стандартна версия;
- Java Virtual Machine – стандартна версия;
- Графичен потребителски интерфейс;
- TCP/IP мрежов интерфейс;
- Работа с XML;
- Работа с файловата система;
- Интерфейс към платформен (native) код (JNI);
- Интерфейс за работа с бази данни (JDBC);
- Отдалечено извикване на методи (RMI-IIOP);
- 2D/3D графични библиотеки;
- Библиотеки за сигурност;
- Инструменти с общо предназначение;
- Много други.

Повечето от тези теми надхвърлят обхвата на настоящата книга и няма да бъдат разглеждани в подробности.

Java Platform Enterprice Edition (Java EE)

Доскоро известна повече като J2EE, тази разновидност на платформата е предназначена за големи приложения с корпоративен характер, най-често разработени да бъдат използвани от хиляди клиенти. Java EE съдържа всичко от Java SE, но го обогатява с още библиотеки и технологии.

Обсегът на Java EE далеч надхвърля обхвата на настоящата книга, затова само ще изброим най-основните технологии, включени в нея.

- HTTP сървър + Servlet контейнер – за разработка на уеб приложения;
- EJB контейнер – за разработка на преизползваеми компоненти за отдалечено извикване;
- Предаване на съобщения (JMS);
- Повишена сигурност;
- Уеб услуги.

Java Platofrm Micro Edition (Java ME)

Доскоро известна повече като J2ME, това е ограничена версия на стандартната с по-малко възможности, предназначена за използване в малки и ограничени откъм ресурси хардуерни устройства. Най-често това са мобилни телефони, персонални помощници (PDA) и дори домакински уреди като печки хладилници и телевизори.

За да постигне задоволителна производителност при тези скромни възможности на устройствата, Java ME опростява и ограничава възможностите на Java SE понякога до съвсем малък набор от базови операции. Това се отразява както на компилатора, така и на JMV.

Java технологиите

Въпреки своята големина и изчерпателност Java платформата не покрива всички възможни теми. Съществуват множество независими производители на софтуер, които разширяват и допълват това, което се предлага от Java. Разширенията са програми, достъпни за преизползване от други Java програми. Преизползването на програмен код съществено улеснява и опростява програмирането. Ние като програмисти използваме наготово написани неща и така само с няколко класа можем да напишем сравнително сложна програма.

Да вземем за пример писането на програма, която визуализира данни под формата на графики и диаграми. Можем да вземем библиотека на писа на Java, която рисува самите графики. Всичко, от което се нуждаем, е да подадем правилните входни данни и библиотеката ще изрисува графиките вместо нас. Много е удобно.

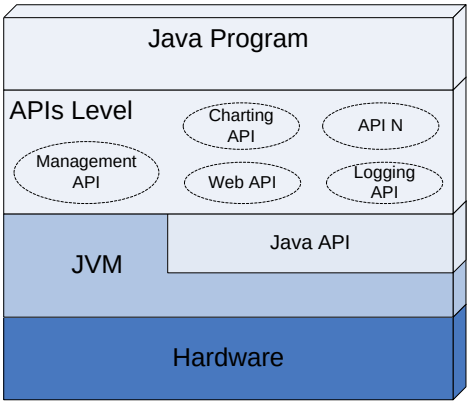
Повечето разширения се използват като инструменти, защото са сравнително прости. Съществуват и разширения, които имат сложна структура и вътрешни зависимости и наподобяват не прости инструменти, а сложни техники, и е по-коректно да се нарекат **технологии**. Съществуват множество Java технологии с различни области на приложение. Типичен пример са уеб технологиите, позволяващи бързо и лесно да се пишат динамични уеб приложения.

Съществуват технологии, които поради качествата си, започват да се използват масово от програмистите. След като се утвърдят такива технологии, те се включват и стават част от самата Java платформата.

Application Programming Interface (API)

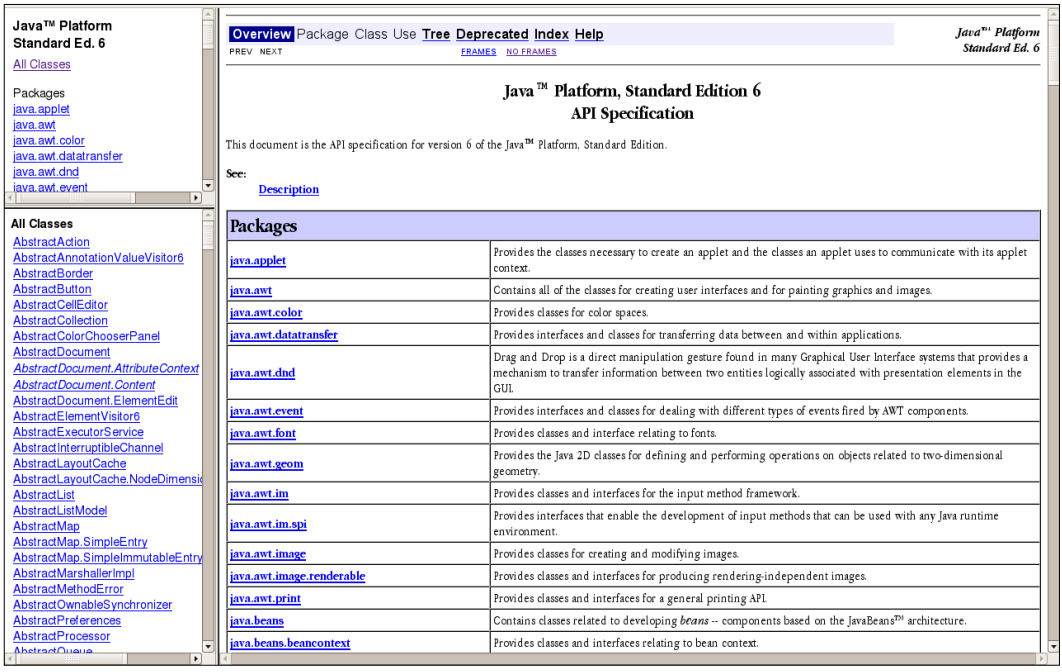
Всеки Java инструмент или технология се използва, като се създават обекти и се викат техни методи. Наборът от публични класове и методи, които са достъпни за употреба от програмистите и се предоставят от технологиите, се нарича **Application Programming Interface** или просто **API**. За пример можем да дадем самия Java API, който е набор от Java класове, разширяващи възможностите на езика, добавяйки функционалност от високо ниво. Всички Java технологии предоставят публичен API. Много

често за самите технологии се говори просто като за API, предоставящ определена функционалност, като например API за работа с файлове, уеб API и т.н. Голяма част от съвременния софтуер използва множество видове API, обособени като отделно ниво от приложението.



Java API документацията

Много често се налага да се документира един API, защото той съдържа множество пакети и класове. Класовете съдържат методи и параметри, смисълът на които не е очевиден и трябва да бъде обяснен. Съществуват вътрешни зависимости между отделните класове и пакети и за правилната им употреба е необходимо разяснение.



Java API документацията е стандартен HTML формат, позволяващ да се опишат пакетите, класовете, методите и свойствата, съдържащи се в дадена

библиотека или инструмент. Тя се пише от програмистите, разработили библиотеката за програмистите, които я използват.

Типичен пример е Java™ Platform, Standard Edition 6 API Specification (http://java.sun.com/javase/6/docs/api/). Тук можем да намерим подробно описание на всичко от стандартната версия на Java платформата, включително препратки към статии, свързани с темата.

Какво ви трябва, за да програмирате на Java?

Базовите изисквания, за да можете да програмирате на Java са: първо - инсталирана Java платформа и второ - текстов редактор. Текстовият редактор служи за създаване и редактиране на Java кода, а за компилиране и изпълнение се нуждаем от самата Java.

Java дистрибуции JDK и JRE

Java SE, Java EE и Java ME са налични за инсталация за повечето съвременни операционни системи в това число Windows, Linux, Mac OS X, Solaris, AIX и др.

За да инсталирате Java SE на настолния си компютър, трябва да изберете подходящата дистрибуция. Съществуват две основни дистрибуции:

- Java Development Kit (JDK);
- Java Runtime Environment (JRE).

Като програмисти на Java ние се нуждаем от JDK. JDK включва в себе си както виртуална машина, така и Java компилатор, а също и множество помощни инструменти. JRE включва в себе си единствено виртуална машина и някои от нейните инструменти се явяват орязана версия на JDK. JRE се използва от хора, които нямат намерение да пишат програми на Java, а искат само да могат да изпълняват вече готови такива. Понякога JDK може да се срещне и като Java Standard Development Kit или Java SDK.

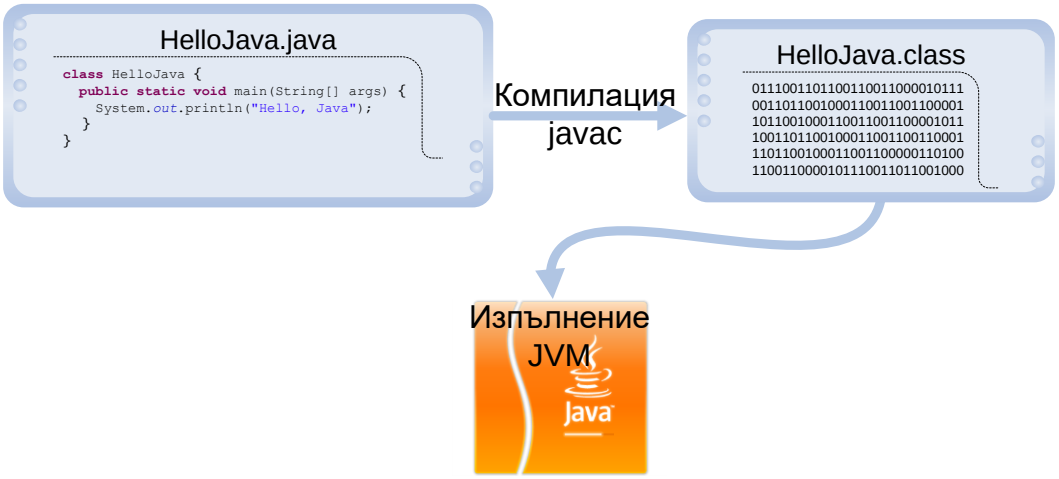
Текстов редактор

Служи за писане на изходния код на програмата и за записването му във файл. След това кодът се компилира и изпълнява.

Компилация и изпълнение на Java програми

Дойде време да приложим на практика вече разгледания теоретично пример на проста програма, написана на Java. Нека изпълним нашия пример. За целта трябва да направим следното:

- Да създадем файл с име **`HelloJava.java`**;
- Да запишем примерната програма във файла;
- Да компилираме **`HelloJava.java`** до файл **`HelloJava.class`**;
- Да подадем **`HelloJava.class`** на JVM за изпълнение.



А сега, нека да го направим на компютъра!

Не забравяйте преди започването с примера, да инсталирате JDK на компютъра си! В противен случай няма да можете да компилирате и да изпълните програмата.

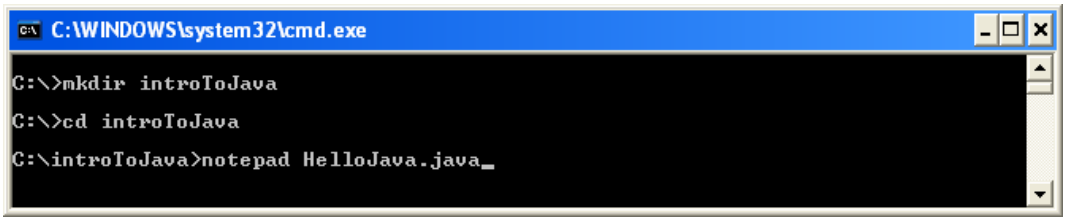
Горните стъпки варират на различните операционни системи и затова ще ги разделим на две. Първо ще реализираме стъпките на Windows, а след това на Solaris и Linux. Стъпките за Solaris и Linux са еднакви, затова ги обединяваме. Всички операции ще бъдат извършени от командния ред (конзолата). Първите две стъпки - създаване на файл и записване на програмата в него – могат да бъдат пропуснати, ако използваме примерния файл:

HelloJava.java

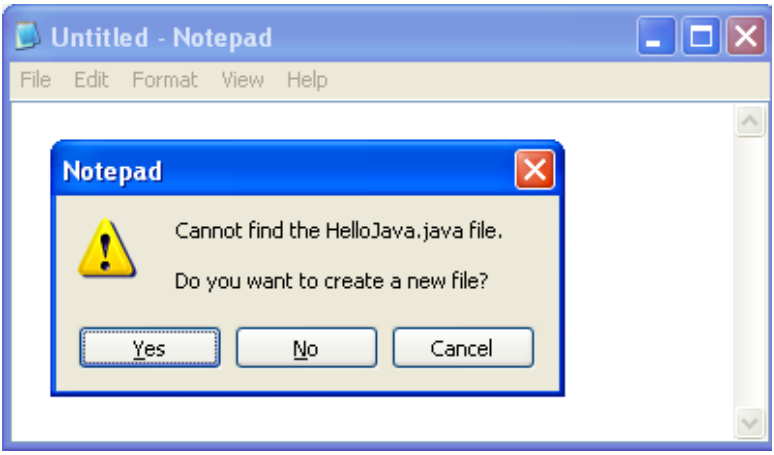
```
class HelloJava {
   public static void main(String[] arguments) {
      System.out.println("Hello, Java");
   }
}
```

Компилиране на Java програми под Windows

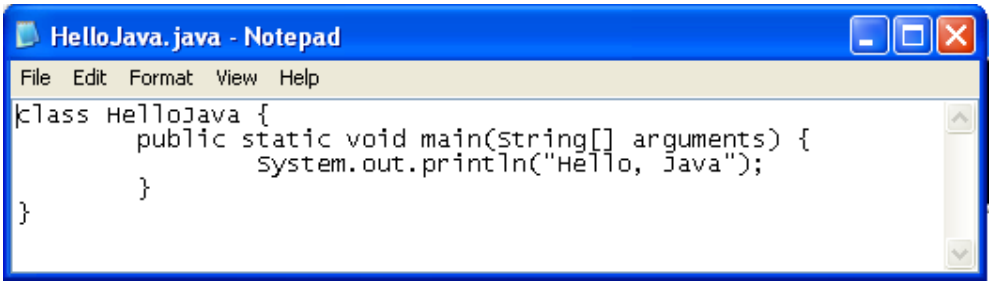
Нека създадем директория, в която ще експериментираме.



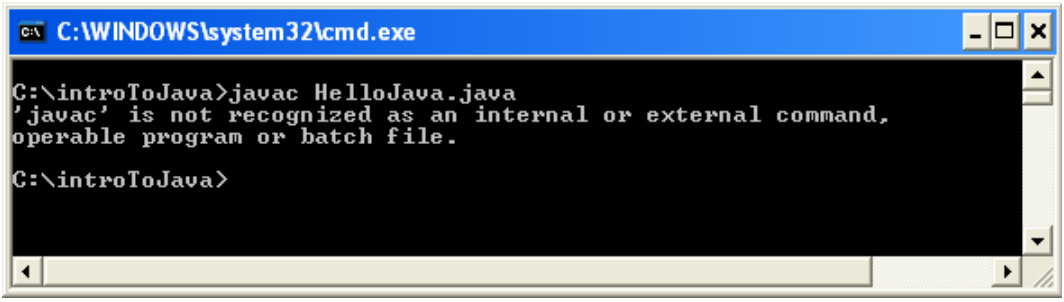
Директорията се казва **introToJava** и се намира в **C:**. Променяме текущата директория на **C:\introToJava** и създаваме нов файл **HelloJava.java,** като за целта използваме вградения текстов редактор Notepad.



На въпроса дали искаме да бъде създаден нов файл, защото такъв в момента липсва, отговаряме с "Yes". Следващото необходимо нещо е да препишем програмата или просто да прехвърлим текста чрез копиране.



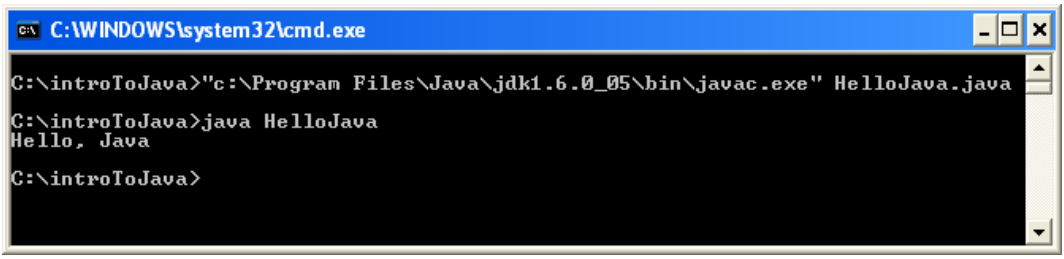
Записваме чрез [Ctrl-S] и затваряме с [Alt-F4]. Вече имаме изходния код на програмата, записан като файл **C:\introToJava\HelloJava.java**. Остава да компилираме и изпълним. Компилацията се извършва с компилатора **javac.exe**.



Ето, че получихме грешка – Windows не може да намери изпълним файл **javac**. Това е често срещан проблем, ако сега започваме да работим с Java, и причините за него са:

- Липсваща инсталирана Java дистрибуция;
- Инсталирана неправилна Java дистрибуция – JRE (трябва ни инсталиран JDK);
- Инсталиран JDK, но **JAVA_HOME/bin** директорията не е в пътя за търсене на изпълними файлове и Windows не намира **javac.exe**, въпреки че то е налично на диска.

Първите два варианта се решават, като се инсталира JDK. Последният се решава посредством използване на пълния път до **javac.exe,** както е показано на картинката долу.



След изпълнението си **javac** излиза без грешки, като произвежда още един файл **C:\introToJava\HelloJava.class**. За да го изпълним, просто го подаваме на виртуалната машина. За стартиране на виртуалната машина използваме изпълнимия файл **java.exe**. Както се вижда на горния екран, използваме командата:

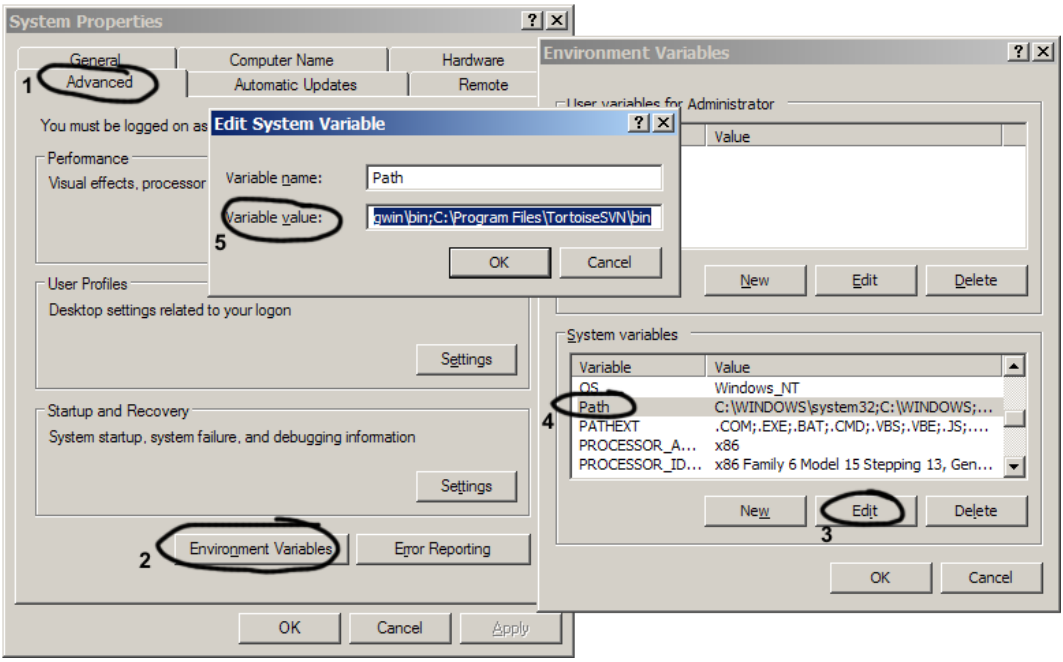
```
java HelloJava
```

Резултатът от изпълнението на нашата първа програма е съобщението "Hello, Java", изписано на конзолата. Не е нещо велико, но е едно добро начало.

Не добавяйте разширението `class`**, когато го подавате за изпълнение от** `java`**! В противен случай ще получите съобщение за грешка.**

Промяна на системните пътища в Windows

Може би ви е досадно всеки път да изписвате пълния път до **javac.exe** или **java.exe**, когато компилирате през конзолата. За да избегнете това, можете да редактирате системните пътища в Windows и след това да затворите конзолата и да я пуснете отново. Промяната на системните пътища в Windows става по следния начин: My Computer --> Properties --> Advanced --> Environment Variables --> System Variables --> Path --> Edit:

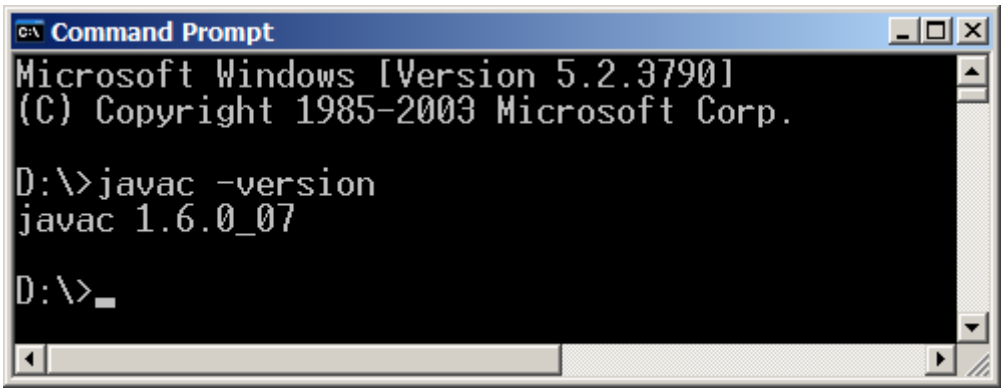


След това добавяме в пътя **bin** директорията на нашата инсталация на JDK. При стандартна инсталация този път би могъл да изглежда примерно така:

```
C:\Program Files\Java\jdk1.6.0_07\bin
```

Пътят в променливата **Path** представлява списък, в който отделните елементи са разделени с точка и запетая. Добавянето на пътя до нашата JDK инсталация става като добавим точка и запетая и самия път. Внимавайте да не изтриете съществуващия списък, защото това ще навреди на компютъра ви и някои програми ще спрат да работят.

Вече сме готови. Трябва да затворим конзолата, защото тя ползва стария път (преди да го променим). След като пуснем нова конзола за команди (**cmd.exe**), ще можем да компилираме конзолно без да изписваме пълния път до компилатора на Java. Ето пример, в който извикваме Java компилатора от конзолата и искаме от него да изпише версията си:



Компилиране на Java програми под Linux и Solaris

Да направим преразказ в картинки на горното, този път за Linux и Solaris операционни системи. Стъпките са аналогични.

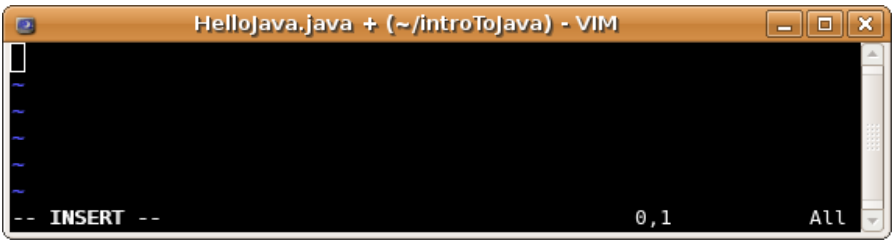
Не забравяйте да инсталирате JDK, преди да се захванете с настоящия пример. В противен случай ще получите съобщения за грешки.

При различните Linux и UNIX дистрибуции инсталирането на JDK става по различен начин и няма общо решение, което работи навсякъде. Четете в документацията на вашата дистрибуция с коя пакетна система работи тя, за да разберете съответната команда за инсталиране (**`rpm -i`**, **`yum install`**, **`apt-get install`**, …) и кой е съответният пакет. Например на Debian и Ubuntu Linux инсталирането на JDK става с командата:

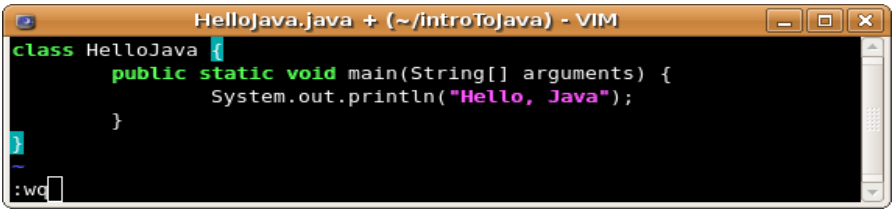
```
apt-get install sun-java6-jdk
```

Сега считаме, че имаме инсталиран JDK. Нека създадем директория, в която ще експериментираме. Директорията се казва **`introToJava`** и се намира в домашната директория на потребителя. Променяме текущата директория на **`~/introToJava`** и създаваме нов файл **`HelloJava.java`**, като за целта използваме вградения текстов редактор **`vi`**.

Влизаме в решим на въвеждане, като натискаме "i":

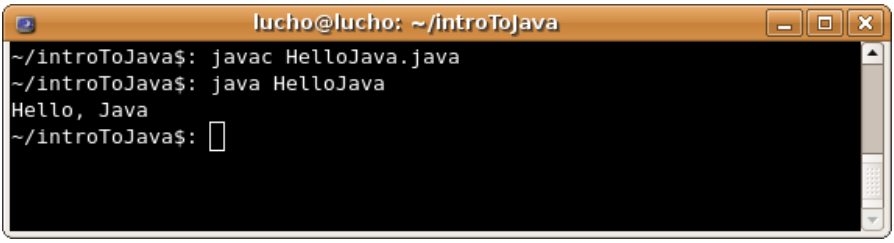


Въвеждаме или прехвърляме нашата програма с копиране. Записваме, като натискаме Esc, за да излезем от режим на въвеждане, след което пишем ":wq" и натискаме Enter:



Всъщност, можем да използваме и друг текстов редактор вместо **vi**, защото, ако за първи път виждате **vi**, много трудно ще свикнете да работите с него. Вместо **vi** можете да използвате по-дружелюбните редактори **mcedit** или **pico** (които за преди това вероятно ще трябва да инсталирате, защото се разпространяват като допълнителни пакети).

Вече имаме файла **~/introToJava/HelloJava.java** и остава да го компилираме и изпълним. Компилацията се извършва с компилатора **javac**:



След изпълнението си **javac** излиза без грешки, като произвежда още един файл **~/introToJava/HelloJava.class**. За да го изпълним, просто го подаваме на виртуалната машина. За стартиране на виртуалната машина използваме изпълнимия файл **java** (който би трябвало да е достъпен от системния път). Както се вижда на горния екран, използваме командата:

```
java HelloJava
```

Резултатът от изпълнението на нашата първа програма е съобщението "Hello, Java", изписано на конзолата.

Средата за разработка Eclipse

До момента разгледахме как се компилират и изпълняват Java програми с конзолата. Разбира се, има и по-лесен начин – чрез използване на среда за разработка, която изпълнява всички команди, които видяхме вместо вас. Нека разгледаме как се работи със среди за разработка и какво ни помагат те, за да си вършим по-лесно работата.

Интегрирани среди за разработка

В горните примери разгледахме компилация и изпълнение на програма от един единствен файл. Обикновено програмите са съставени от много файлове, понякога дори десетки хиляди. Писането с текстов редактор, компилирането и изпълнението на една програма от командния ред е сравнително проста работа, но да направим това за голям проект, може да се окаже сложно и трудоемко занимание. За намаляване на сложността, улесняване на писането, компилирането, изпълнението и интегрирането им в един единствен инструмент, съществуват визуални приложения, наречени **интегрирани среди за разработка** (Integrated Development Environment, IDE). Средите за разработка най-често предлагат множество допълнения към основните функции, като дебъгване, пускане на unit тестове, проверка на често срещани грешки, контрол на версиите и т.н.

Какво е Eclipse?

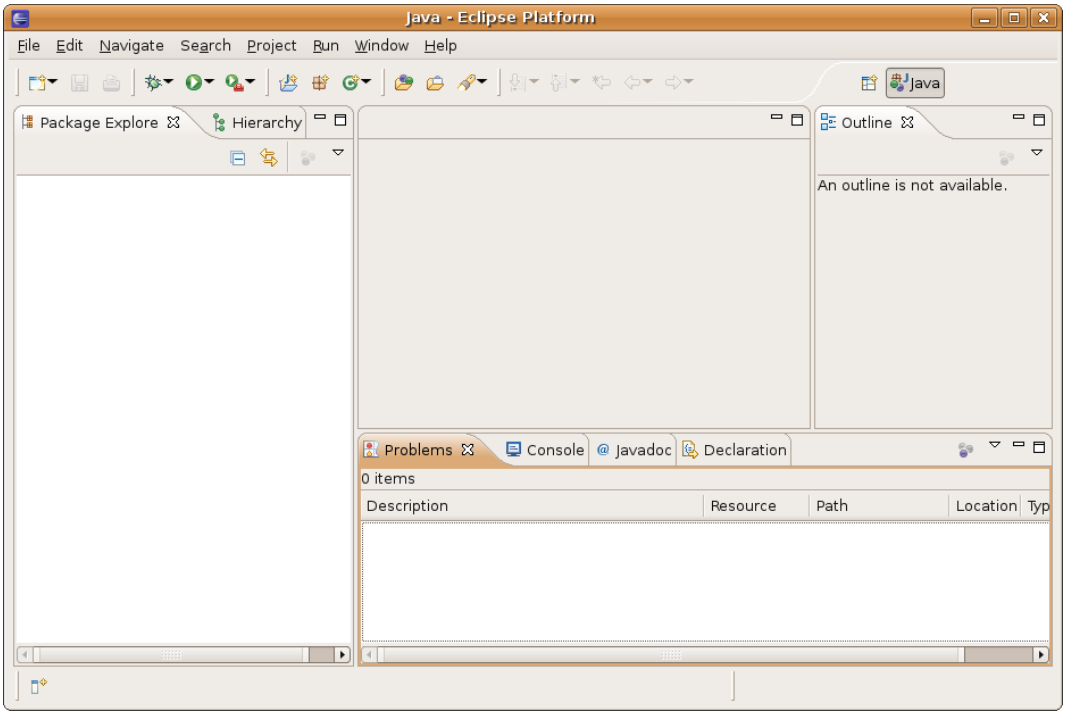
Eclipse (www.eclipse.org) е мощна интегрирана среда за разработка на софтуерни приложения. Eclipse е проект с отворен код и предлага инструменти за всички етапи от софтуерния жизнен цикъл. Тук ще разгледаме най-важните функции – тези, свързани със самото програмиране – писането, компилирането, изпълнението и дебъгването на програми.

Преди да преминем към примера, нека разгледаме малко по-подробно структурата на визуалния интерфейс на Eclipse. Основна съставна част са **перспективите**. Всяка перспектива съдържа един или повече **визуализатори (views)**. Всеки визуализатор реализира различна функция, свързана с улесняване на програмирането. Да разгледаме най-използваната перспектива – Java перспективата. Тя съдържа:

- Package Explorer – тук могат да се видят всички класове, които съставят текущия проект на програма;
- Problems – показва грешките при компилация;
- Outline – показва свойствата и поведението (методите) на селектиран Java клас;
- Console – показва стандартния изход, когато стартираме програма.

Освен визуализатори в средата на работното пространство стоят **редакторите** на Java файлове. На примера по-долу няма нито един отворен файл

за редактиране и затова пространството в средата е празно. В Eclipse можем да редактираме едновременно произволен брой файлове.



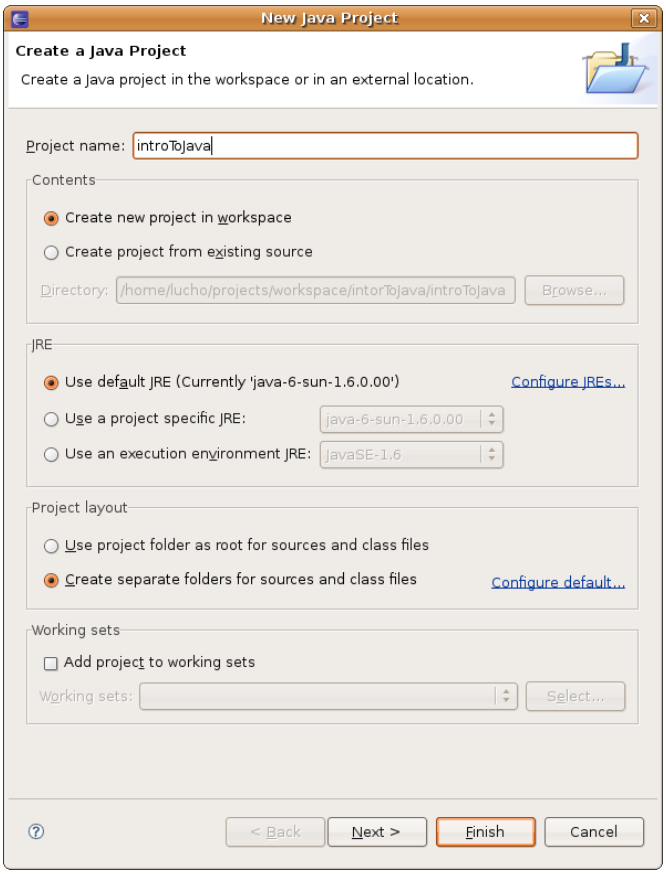
Съществуват още редица визуализатори с помощно предназначение, които няма да разглеждаме тук.

Създаване на нов Java проект

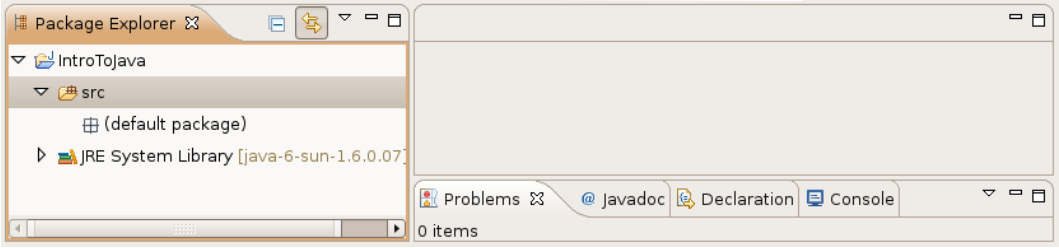
Преди да направим каквото и да било в Eclipse, трябва да създадем проект за него. Проектът логически групира множество файлове, предназначени да реализират произволна програма. За всяка програма е добре да се създава отделен проект.

Проект се създава чрез следване на следните стъпки:

- File -> New -> Project;
- Избираме "Java Project";
- Въвеждаме името на проекта ни - примерно "introToJava";
- Натискаме "Finish".



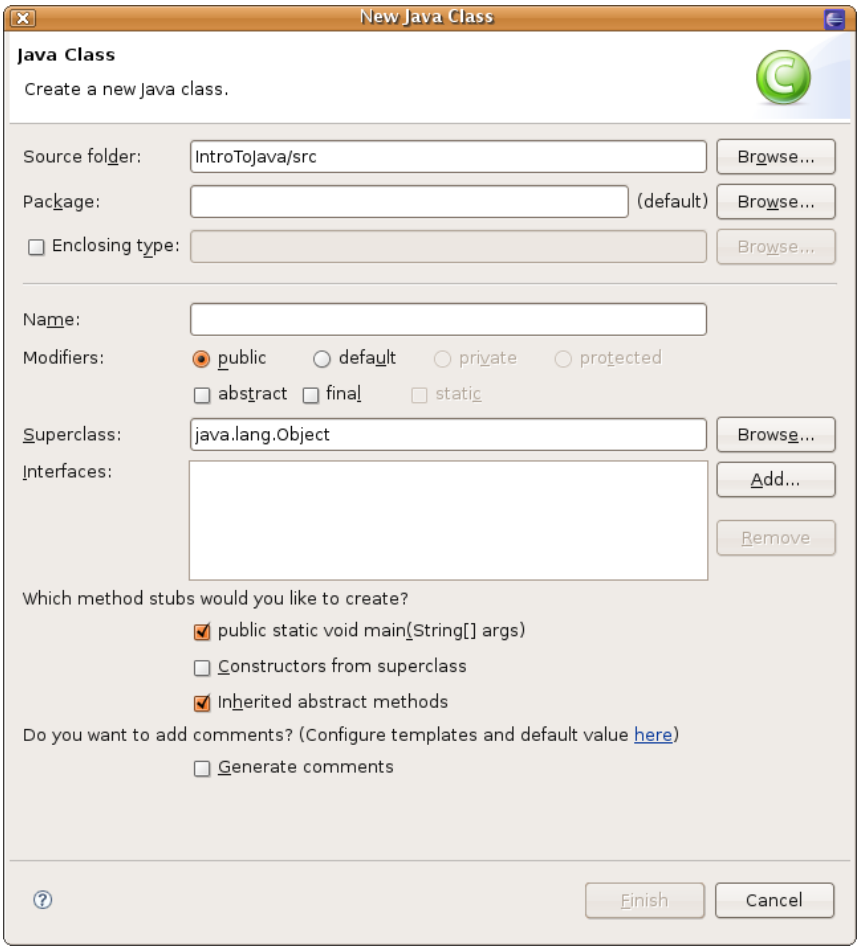
Новосъзданеният проект се показва в Package Explorer.



Вече можем да добавяме класове към проекта.

Създаване на нов клас

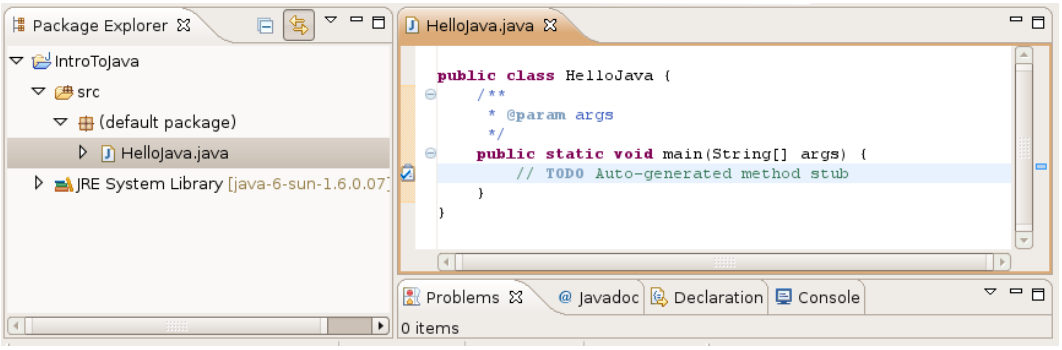
Eclipse предлага помощен прозорец за създаването на нови класове:



За да създадем нов клас:

- Избираме File-> New -> Class;
- В помощния прозорец попълваме името на класа - примерно "HelloJava";
- Селектираме опцията "public static void main(String[] args)" – това автоматично ще генерира **`main()`** метод за нас;
- Натискаме бутона [Finish].

Новосъзданеният клас се появява в Package Explorer и се отваря нов редактор, където може да въведем съдържанието на новия клас. Eclipse е генерирал тялото на новия клас, метод **`main()`** и няколко помощни коментара:

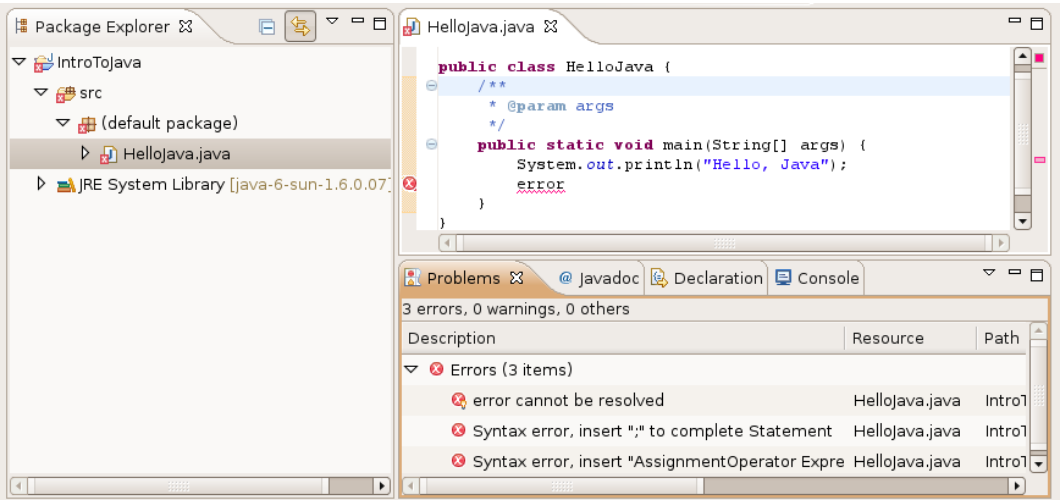


Въвеждаме съдържанието на **main()** метода и сме готови за компилация и изпълнение:

Компилиране на сорс кода

Процесът на компилация в Eclipse включва няколко стъпки:

- Проверка за синтактични грешки;



- Проверка за грешки на типовете;
- Преобразуване на Java кода в изпълними bytecode инструкции.

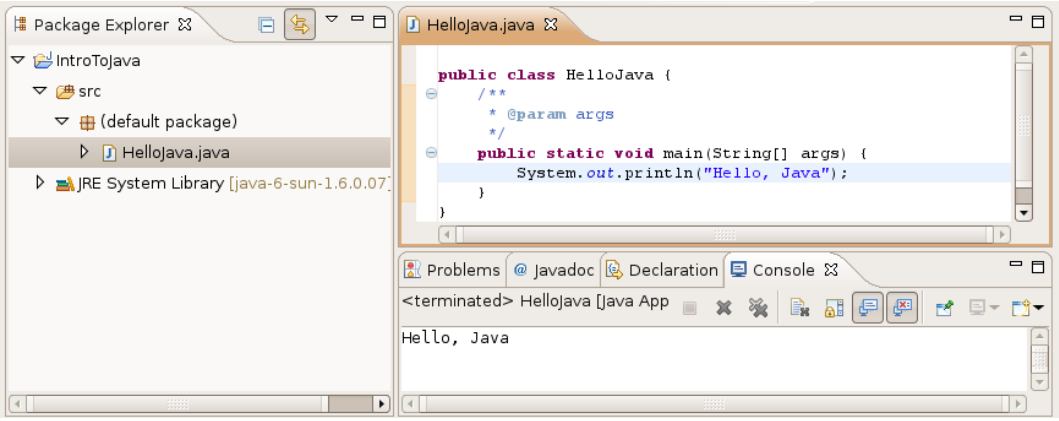
За да компилираме класа си в Eclipse, всъщност не е нужно да правим нищо допълнително. Компилацията се извършва още докато пишем. Намерените грешки се подчертават в червено за заостряне на вниманието. Когато запишем промените, компилацията отново се стартира автоматично. Намерените грешки се показват във визуализатора "Problems".

Ако в проекта ни има поне една грешка, то тя се отбелязва с малък червен "x" в Package Explorer. Ако разгърнем проекта на ниво пакети и класове, тези, които съдържат грешки, се отбелязват по същия начин. Ако щракнем двойно на някоя от грешките в Problems Eclipse, ни прехвърля автоматично на мястото в кода, където е възникнала грешката.

Стартиране на проекта

За да стартираме проекта, избираме Run -> Run As -> Java Application или клавишната комбинация Ctrl+Alt+X J (Натискаме Ctrl, Alt и X заедно, след което натискаме J).

Програмата се стартира и резултатът се изписва във визуализатора "Console":



Не всички типове проекти могат да се изпълнят. За да се изпълни Java проект, е необходимо той да съдържа поне един клас с `main()` метод.

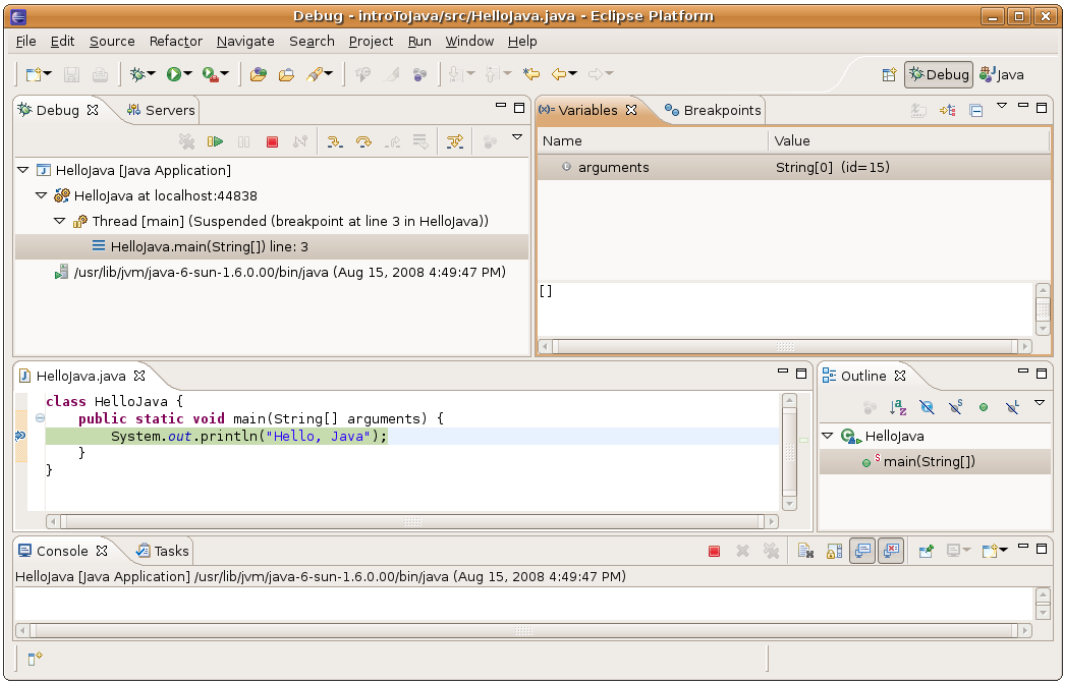
Дебъгване на програмата

Когато програмата ни съдържа грешки, известни още като **бъгове**, трябва да намерим и отстраним тези грешки. Трябва да **дебъгнем** програмата. Процесът на дебъгване включва:

- Забелязване на бъговете;
- Намиране на кода, който причинява проблемите;
- Оправяне на кода, така че програмата да работи правилно;
- Тестване, за да се убедим, че програмата работи правилно след корекциите.

Процесът може да се повтори няколко пъти и е продължителен.

Eclipse предлага отделна перспектива за дебъгване:



След като сме забелязали проблема в програмата си, ние трябва да намерим кода, който го създава. Eclipse може да ни помогне с това, като ни позволи да проверим дали всичко работи, както е планирано.

За да спрем изпълнението на програмата, в тези места поставяме **точки на прекъсване**, известни още като **стопери (breakpoints)**. Стоперът е асоцииран към ред от програмата ни. Програмата спира изпълнението си на тези редове и позволява постъпково изпълнение на останалите редове. На всяка стъпка може да проверяваме и дори променяме съдържанието на текущите променливи.

Дебъгването е един вид изпълнение на програмата на забавен кадър. То ни дава възможност по-лесно да вникнем в детайлите и да видим къде са грешките.

Упражнения

1. Да се намери описанието на класа System в стандартната Java API документация.
2. Да се намери описанието на метода **`System.out.println()`**, с различните негови параметри, в стандартната Java API документация.
3. Да се компилира и изпълни примерната програма от тази глава през командния ред (конзолата) и с помощта на Eclipse.

4. Да се модифицира примерната програма, така че да изписва различно поздравление, например "Добър ден!".

Решения и упътвания

1. Използвайте http://java.sun.com/javase/6/docs/api/ и потърсете класа **`System`**.
2. В описанието на класа **`System`** потърсете полето **`out`**, след което намерете метода **`println()`**.
3. Използвайте инструкциите, дадени в текста на тази глава.
4. Използвайте инструкциите, дадени в текста на тази глава.

Глава 2. Примитивни типове и променливи

Автор

Христо Тодоров

Светлин Наков

В тази тема...

В настоящата тема ще разгледаме примитивните типове и променливи в Java - какво представляват и как се работи с тях. Първо ще се спрем на типовете данни – целочислени типове, реални типове с плаваща запетая, булев тип, символен тип, обектен тип и стрингов тип. Ще продължим с това какво е променлива, какви са нейните характеристики, как се декларира, как се присвоява стойност и какво е инициализация на променлива. Ще се запознаем и с другото наименование на променливата, а по-точно – "идентификатор". Към края на главата ще се спрем на литералите. Ще разберем какво представляват, какви видове са и на последно място ще срещнем упражнения приготвени за нас читателите, чиято цел е да затвърдим знанията, които ще придобием прочитайки главата.

Какво е променлива?

Една типична програма използва различни стойности, които се променят по време на нейното изпълнение. Например, създаваме програма, която извършва пресмятания. Стойностите, въведени от един потребител, ще бъдат очевидно различни от тези, въведени от друг потребител. Това означава, че когато създаваме програмата, ние не знаем всички възможни стойности, които ще бъдат въведени в нея. Това от своя страна изисква да можем да обработим стойностите, които потребителите евентуално биха въвели.

Нека създадем програма, чиято цел е да извършва пресмятания. Когато потребителят въведе нова стойност, която ще участва в процеса на пресмятане, ние можем я да съхраним (временно) в паметта на нашия компютър. Стойностите в тази част на паметта се променят регулярно. Това е довело до наименованието им – променливи.

Типове данни

Тип данни представлява съвкупност от стойности, които имат еднакви характеристики.

Характеристики

Типовете данни се характеризират с:

- Име;
- Размер (колко памет заемат);
- Стойност по подразбиране (**default value**).

Видове

Типовете данни се разделят на следните видове:

- Целочислени типове – **`byte`**, **`short`**, **`int`**, **`long`**;
- Реални типове с плаваща запетая – **`float`** и **`double`**;
- Булев тип – **`boolean`**;
- Символен тип – **`char`**;
- Обектен тип – **`Object`**;
- Символни низове – **`String`**.

В следната таблица можем да видим изброените по-горе типове данни (**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**, **`Object`** и **`String`**), включително стойностите им по подразбиране и техния обхват:

Тип данни	Стойност по подразбиране	Минимална стойност	Максимална стойност
`byte`	**`0`**	-128	+127
`short`	**`0`**	-32768	+32767
`int`	**`0`**	-2147483648	+2147483647
`long`	**`0L`**	-9223372036854775808	+9223372036854775807
`float`	**`0.0f`**	-3.4E+38	+3.4E+38
`double`	**`0.0d`**	-1.7E+308	+1.7E+308
`boolean`	**`false`**	Възможните стойности са две – **`true`** или **`false`**	
`char`	**`'\u0000'`**	0	+65535
`Object`	**`null`**		
`String`	**`null`**		

Типовете **byte**, **short**, **int**, **long**, **float**, **double**, **boolean** и **char** се наричат **примитивни типове данни**, тъй като са вградени в езика Java на най-ниско ниво.

Типовете **Object** и **String** са изписани с главна буква, тъй като са сложни типове (не са примитивни). Те представляват класове, които са дефинирани чрез средствата на езика Java, а не са част от него самия, а са част от стандартните библиотеки на Java.

Целочислени типове

Целочислените типове отразяват целите числа и биват – **byte**, **short**, **int** и **long**. Нека ги разгледаме един по един в реда, в който ги изброихме.

Първи в нашия списък е целочисленият тип **byte**. Той е 8-битов знаков тип, което означава, че броят на възможните стойности е 2 на степен 8, т.е. 256 възможни положителни и отрицателни стойности общо. Стой–ността по подразбиране е числото 0. Минималната стойност, която заема, е -128, а максималната +127.

Вторият по ред в нашия списък е целочисленият тип **short**. Той е 16-битов знаков тип. В параграфа за типа **byte** по-горе изяснихме какво означава знаков тип и какво значение отдава броят на битовите. Стойността по подразбиране е числото 0. Минималната стойност, която заема е числото -32768, а максималната - +32767.

Следващият целочислен тип, който ще разгледаме е типът **int**. Той е 32-битов целочислен знаков тип. Както виждаме, с нарастването на битовете нарастват и възможните стойности, които даден тип може да заема. Стойността по подразбиране е числото 0. Минималната стойност, която заема е -2 147 483 648, а максималната +2 147 483 647.

Последният целочислен тип, който ни предстои да разгледаме, е типът **long**. Той е 64-битово цяло число със знак със стойност по подразбиране **0L**. Знакът **L** се указва, че числото е от тип long (иначе се подразбира **int**). Минималната стойност, която типът **long** заема, е -9 223 372 036 854 775 808, а максималната +9 223 372 036 854 775 807.

Когато можем да използваме типът byte вместо типа int или long, не се колебайте да използвате byte. Това ще спести от заеманата в компютъра памет.

Целочислени типове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни целочислени типове, инициализираме ги и ги отпечатваме на конзолата. Какво представлява деклариране и инициализация на променлива, можем да прочетем по-долу в секциите, наименувани "Деклариране на променливи" и "Инициализация на променливи".

```
// Declare some variables
byte centuries = 20;
short years = 2000;
int days = 730480;
long hours = 17531520;
// Print the result on the console
System.out.println(centuries + " centuries is " + years +
   " years, or " + days + " days, or " + hours + " hours.");
```

В разгледания по-горе пример демонстрираме използването на целочислените типове. За малки числа използваме типът **byte**, а за много големи – целочисленият тип **long**.

Нека видим резултата от горния фрагмент код отдолу.

```
20 centuries is 2000 years, or 730480 days, or 17531520 hours.
```

Реални типове с плаваща запетая

Реалните типове с плаваща запетая представляват реалните числа, които познаваме, и биват – **float** и **double**. Нека подходим както с целочислените типове и да ги разгледаме един след друг, за да разберем какви са разликите между двата типа и защо при изчисления понякога се държат обратно на очакваното.

Първи в нашия списък е 32-битовият реален тип с плаваща запетая **float**. Стойността по подразбиране е **0.0f** или **0.0F** (двете са еквиваленти). Символът "**f**" накрая указва изрично, че числото е от тип **float** (защото по подразбиране всички реални числа са от тип **double**). Разглежданият тип има точност от 6 до 9 десетични знака (останалите се губят). Минималната стойност, която може да заема, е **-3.4E+38**, а максималната е **+3.4E+38**.

Втория реален тип с плаваща запетая, който ще разгледаме, е типът **double**. Той е 64-битов тип със стойност по подразбиране **0.0d** или **0.0D**. Разглежданият тип има точност от 15 до 17 десетични знака. Минималната стойност, която представя, е **-1.7E+308**, а максималната е **+1.7E+308**.

Реални типове – пример

Ето един пример за деклариране променливи от тип число с плаваща запетая и присвояване на стойности за тях:

```
// Declare some variables
float floatPI = 3.14f;
double doublePI = 3.14;
```

Точност на реалните типове

Разгледахме два реални типа – **float** и **double**. Освен с броя на възможните стойности, които могат да заемат, се различават и с точността им (броя десетични цифри, които запазват). Първият тип има точност от 6 до 9 знака, а вторият – от 15 до 17 знака.

Точност на реални типове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни реални типове, инициализираме ги и ги отпечатваме на конзолата. Целта на примера е да онагледим разликата в точността на двата реални типа – **float** и **double**.

```
// Declare some variables
float floatPI = 3.141592653589793238f;
double doublePI = 3.141592653589793238;
// Print the result on the console
System.out.println("Float PI is: " + floatPI);
System.out.println("Double PI is: " + doublePI);
```

Нека видим резултата от горния фрагмент код отдолу, за да ни се изясни какво е това точност при реалните типове:

```
Float PI is: 3.1415927
Double PI is: 3.141592653589793
```

В примера по-горе декларирахме две променливи, една от тип **double** и една от тип **float**, инициализираме ги и ги отпечатваме. Виждаме, че числото пи, декларирано от тип **float**, е закръглено на 7-ми знак, а от тип **double** – на 15-ти знак. Изводът, който бихме си направили, е че реалният тип **double** е с доста по-голяма точност и ако ни е необходима голяма точност след десетичния знак, ще използваме него.

За представянето на реалните типове

Реалните числа в Java се представят като **числа с плаваща запетая** (съгласно стандарта **IEEE 754**) и се състоят от три компонента: **знак** (1 или -1), **мантиса** (значещи цифри) и **експонента** (скала на отместване), като стойността се изчислява по сложна формула. Мантисата съхранява значещите цифри на числото и има капацитет до 9 десетични цифри при float и до 17 десетични цифри при double. При използване на повече цифри, те се губят и стават нули. Загуба на точност се получава не само при голям брой цифри (както ще видим малко по-късно). Експонентата описва на коя позиция се намира десетичната точка. Благодарение на нея числата с плаваща запетая могат да бъдат много близки до нулата (до $1.4 * 10^{-45}$ за типа float) и много големи (до $3.4 * 10^{38}$ за типа float), но не всяко число има точно представяне.

Повече за представянето на числата ще научите в главата "Бройни системи".

При представянето на дадено реално число във формата с плаваща запетая много често пъти се губи точност, най-малкото, защото всички реални числа са безкрайно много, а всички числа, които могат да се представят в краен брой битове (32 или 64) са краен брой. Например числото 0.1 (една десета) няма точно представяне като число с плаваща запетая (т.е. в типовете float и double) и се представя приблизително.

Не всички реални числа имат точно представяне в типовете `float` и `double`! Например числото 0.1 се представя закръглено в типа `float` като 0.099999994.

За съжаление в Java няма примитивен тип данни, който съхранява реални числа с фиксирана запетая (при които няма такава загуба на точност).

Грешки при пресмятания с реални типове

При пресмятания с реални типове данни може да наблюдаваме странно поведение. Причината за това е невъзможността някои реални числа да се представят точно в двоичен вид. Примери за такива числа са 1/3, 2/7 и други.

Грешки при пресмятания с реални типове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни реални типове, инициализираме ги, сравняваме ги и отпечатваме резултата на конзолата. Дали двете променливи имат еднаква стойност, ще видим, ако погледнем по-долу:

```
// Declare some variables
float sum =
  0.1f + 0.1f + 0.1f + 0.1f + 0.1f +
  0.1f + 0.1f + 0.1f + 0.1f + 0.1f;
float num = 1.0f;
// Is sum equal to num
boolean equal = (num == sum);
// Print the result ot the console
System.out.println(
  "num = " + num + " sum = " + sum + " equal = " + equal);
```

Нека видим резултата от горния фрагмент код отдолу.

```
num = 1.0 sum = 1.0000001 equal = false
```

От примера можем да заключим, че сумирането на **`0.1f`** десет пъти не е равно на числото **`1.0f`**. Причината за това е, че числото **`0.1f`** е всъщност закръглено до **`0.099999994f`** при записването му в тип **`float`**. Знакът **`f`**, както

вече обяснихме, задава стойност (литерал) от тип **float** и за него можем да прочетем повече в секцията "Реални литерали".

Булев тип

Булевия тип се декларира с ключовата дума **boolean**. Има две стойности, които може да приема – **true** и **false**. Стойността по подразбиране е **false**. Използва се най-често в логически изрази.

Булев тип – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни типове, инициализираме ги, сравняваме ги и отпечатваме резултата на конзолата. Дали двете променливи имат еднаква стойност ще видим, ако погледнем по-долу:

```
// Declare some variables
int a = 1;
int b = 2;
// Which one is greater?
boolean greaterAB = (a > b);
// Is it equal to 1?
boolean equalA1 = (a == 1);
// Print the result on the console
if (greaterAB) {
   System.out.println("A > B");
} else {
   System.out.println("A <= B");
}
System.out.println("greaterAB = " + greaterAB);
System.out.println("equalA1 = " + equalA1);
```

Нека видим резултата от горния фрагмент код:

```
A <= B
greaterAB = false
equalA1 = true
```

В примера декларираме две променливи от тип **int**, сравняваме ги и резултата го присвояваме на променливата от булев тип **greaterAB**. Аналогично за променливата **equalA1**. Ако променливата **greaterAB** е **true**, на конзолата се отпечатва **A > B**, в противен случай **B > A**.

Символен тип

Символният тип представя символна информация. Декларира се с ключовата дума **char**. На всеки символ отговаря цяло число. За да илюстрираме казаното за символния тип, нека разгледаме примерите по-долу.

Символен тип – пример

Нека разгледаме един пример, в който декларираме една променлива от тип **char**, инициализираме я със стойност **'a'**, **'b'** и **'A'** и респективно отпечатваме резултата на конзолата:

```
// Declare a variable
char symbol = 'a';
// Print the result ot the console
System.out.println(
  "The code of '" + symbol + "' is: " + (int) symbol);
symbol = 'b';
System.out.println(
  "The code of '" + symbol + "' is: " + (int) symbol);
symbol = 'A';
System.out.println(
  "The code of '" + symbol + "' is: " + (int) symbol);
```

Нека видим резултата от горния фрагмент код отдолу:

```
The code of 'a' is: 97
The code of 'b' is: 98
The code of 'A' is: 65
```

Символни низове (стрингове)

Символните низове отразяват поредица от символи. Декларира се с ключовата дума **String**. Стойността по подразбиране е **null**. Стринговете се ограждат в двойни кавички, могат да се конкатенират (долепват един до друг), разделят и други. За повече информация можем да прочетем глава 12 **"Символни низове"**, в която детайлно е обяснено какво е това стринг, за какво служи и как да го използваме.

Символни низове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатия ни символен тип, инициализираме ги и ги отпечатваме на конзолата:

```
// Declare some variables
String firstName = "Ivan";
String lastName = "Ivanov";
String fullName = firstName + " " + lastName;
// Print the result ot the console
System.out.println("Hello, " + firstName + "!");
System.out.println("Your full name is " + fullName + ".");
```

Да видим резултата от горния фрагмент код:

```
Hello, Ivan!
Your full name is Ivan Ivanov.
```

Обектен тип

Обектният тип е специален тип, който се явява родител на всички други типове. Декларира се с ключовата дума **`Object`** и може да приема стойности от всеки друг тип.

Използване на обекти – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатия ни обектен тип, инициализираме ги и ги отпечатваме на конзолата:

```
// Declare variables
Object container = 5;
Object container2 = "Five";
// Print the result ot the console
System.out.println("The value of container is: " + container);
System.out.println("The value of container2 is: " + container2);
```

Нека видим резултата от горния фрагмент код:

```
The value of container is: 5
The value of container2 is: Five.
```

Променливи

След като разгледахме основните типове данни в Java, нека видим как и за какво можем да ги използваме. За да работим с данни, трябва да използваме променливи.

Променливата е контейнер на информация, който може да променя стойността си. Тя осигурява възможност за:

- Запазване на информация;
- Извличане на запазената там информация;
- Модифициране на запазената там информация.

Характеристики на променливите

Променливите се характеризират с:

- Име;
- Тип (на запазената в тях информация);
- Стойност (запазената информация).

Именуване на променлива – правила

Когато искаме компилаторът да задели област в паметта за някаква информация, използвана в програмата ни, трябва да зададем име, което служи като идентификатор и позволява да се реферира нужната ни област от паметта.

Името може да бъде всякакво по наш избор, но трябва да следва определени правила:

- Имената на променливите се образуват от буквите **a**-**z**, **A**-**Z**, цифрите **0**-**9**, както и символите **$** и **_**. По принцип е допустимо да се ползват и букви от кирилицата, но това трябва да се избягва.
- Имената на променливите не може да започват с цифра.
- Имената на променливите не могат да съвпадат със служебна дума (keyword) от езика Java:

abstract	**default**	**if**	**private**	**this**
boolean	**do**	**implements**	**protected**	**throw**
break	**double**	**import**	**public**	**throws**
byte	**else**	**instanceof**	**return**	**transient**
case	**enum**	**int**	**short**	**try**
catch	**extends**	**interface**	**static**	**void**
char	**final**	**long**	**strictfp**	**validate**
class	**finally**	**native**	**super**	**while**
const	**float**	**new**	**switch**	
continue	**for**	**package**	**synchronized**	

Именуване на променливи - примери

Правилно име:

- **name**
- **first_Name**
- **_name1**

Неправилно име (ще доведе до грешка при компилация):

- **1** (цифра)
- **if** (служебна дума)
- **1name** (започва с цифра)

Именуване на променливи – препоръки

Ще дадем някои препоръки за именуване, тъй като не всички позволени от компилатора имена са подходящи за нашите променливи.

- Имената трябва да са описателни – да обясняват за какво служи дадената променлива. Примерно за име на човек подходящо име е **personName**, а неподходящо име е **a37**.
- Трябва да се използват само латински букви.
- В Java е прието променливите да започват винаги с малка буква и всяка следваща дума да започва с главна буква. Примерно: **firstName**, а не **firstname** или **first_name**.
- Името трябва да не е нито много дълго, нито много късо.
- Трябва да се внимава за главни и малки букви, тъй като Java прави разлика между тях.

Ето няколко примера за добре именувани променливи:

- **firstName**
- **age**
- **startIndex**
- **lastNegativeNumberIndex**

Ето няколко примера за лошо именувани променливи (макар и имената да са коректни от гледана точка на компилатора на Java):

- **_firstName**
- **last_name**
- **AGE**
- **Start_Index**
- **lastNegativeNumber_Index**

Променливите трябва да имат име, което обяснява накратко за какво служат. Когато една променлива е именувана с неподходящо име, това силно затруднява четенето на програмата и нейното последващо променяне (след време, когато сме забравили как работи тя).

Стремете се винаги да наименувате променливите с кратки, но достатъчно ясни имена, като следвате винаги правилото, че от името на променливата трябва да става ясно за какво се използва.

Деклариране на променливи

Когато декларираме променлива, ние:

- Задаваме нейния тип;
- Задаваме нейното име (наречено идентификатор);
- Може да дадем начална стойност, но не е задължително.

Ето какъв е синтаксисът за деклариране на променливи:

```
<тип данни><идентификатор> [= <инициализация>]
```

Ето няколко примера за деклариране на променливи и задаване на стойностите им:

```
byte centuries = 20;
short years = 2000;
int days = 730480;
long hours = 17531520;
float floatPI = 3.141592653589793238f;
double doublePI = 3.141592653589793238;
boolean isEmpty = true;
char symbol = 'a';
String firstName = "Ivan";
```

Присвояване на стойност

Присвояването на стойност на променлива представлява задаване на стойност на същата. Тази операция се извършва от оператора за присвояване "=". От лявата страна на оператора се изписва типа на променливата и нейното име, а от дясната страна – стойността.

Присвояване на стойност – примери

Ето няколко примера за присвояване на стойност на променлива:

```
int firstValue = 5;
int secondValue;
int thirdValue;
// Using an already declared variable:
secondValue = firstValue;
// The following cascade calling assigns 3 to firstValue and
// then firstValue to thirdValue, so both variables have
// the value 3 as a result:
// Avoid this!
thirdValue = firstValue = 3;
```

Инициализация на променливи

След като вече се запознахме как се декларира променлива, ще пристъпим към задаване на стойност по време на деклариране. Извършвайки тази операция, ние всъщност инициализираме променливата, задавайки стойност на последната.

Всеки тип данни има стойност по подразбиране (инициализация по подразбиране). В таблицата отдолу може да ги видим.

Тип данни	Стойност по подразбиране	Тип данни	Стойност по подразбиране

byte	0	double	0.0d
short	0	char	'\u0000'
int	0	String	null
long	0L	boolean	false
float	0.0f		

Инициализация на променливи – примери

Ето един пример за инициализация на променливи:

```
// The following assigns empty String to the variable "name":
String name = new String(); // name = ""
// This is how we use a literal expression
float heightInMeters = 1.74f;
// Here we use an already initialized variable
String greeting = "Hello World!";
String message = greeting;
```

Стойностни и референтни типове

Типовете данни в Java са 2 вида: стойностни и референтни.

Стойностните типове (value types) се съхраняват в стека за изпълнение на програмата и съдържат директно стойността си. Стойностни са примитивните числови типове, символният тип и булевият тип: **byte**, **int**, **short**, **long**, **float**, **double**, **char**, **boolean**. Такива променливи заемат 1, 2, 4 или 8 байта в стека. Те се освобождават при излизане от обхват.

Референтните типове (reference types) съхраняват в стека за изпълнение на програмата референция към динамичната памет (т. нар. heap), където се съхранява реалната им стойност. Референцията представлява указател (адрес на клетка от паметта), сочещ реалното местоположение на стойността в динамичната памет. Референцията има тип и може да има като стойност само обекти от своя тип, т.е. тя е типизиран указател. Всички референтни (обектни) типове могат да имат стойност **null**. Това е специална служебна стойност, която означава, че липсва стойност. Референтните типове заделят динамична памет при създаването си и се освобождават по някое време от системата за **почистване на паметта (garbage collector)**, когато тя установи, че вече не се използват от програмата. Тъй като заделянето и освобождаването на памет е бавна операция, може да се каже, че референтните типове са по-бавни от стойностните.

Референтни типове са всички класове, масивите, изброените типове и интерфейсите, например типовете: **Object**, **String**, **Integer**, **byte[]**. С обектите, символните низове, масивите и интерфейсите ще се запознаем в следващите глави на книгата. Засега е достатъчно да знаете, че всички

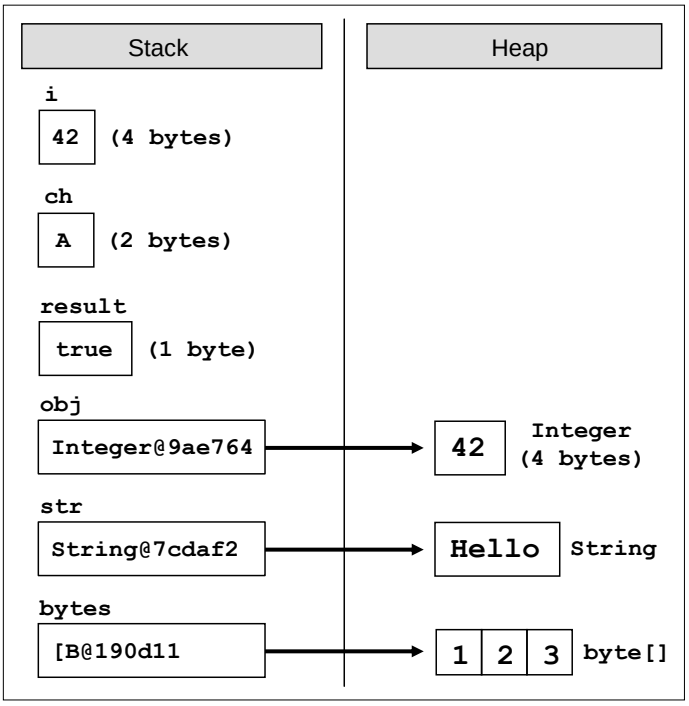
типове, които не са стойностни, са референтни и се разполагат в динамичната памет.

Стойностни и референтни типове и паметта

Нека илюстрираме с един пример как се представят в паметта стойностните и референтните типове. Нека е изпълнен следния програмен код:

```
int i = 42;
char ch = 'A';
boolean result = true;
Object obj = 42;
String str = "Hello";
byte[] bytes = {1, 2, 3};
```

В този момент променливите са разположени в паметта по следния начин:

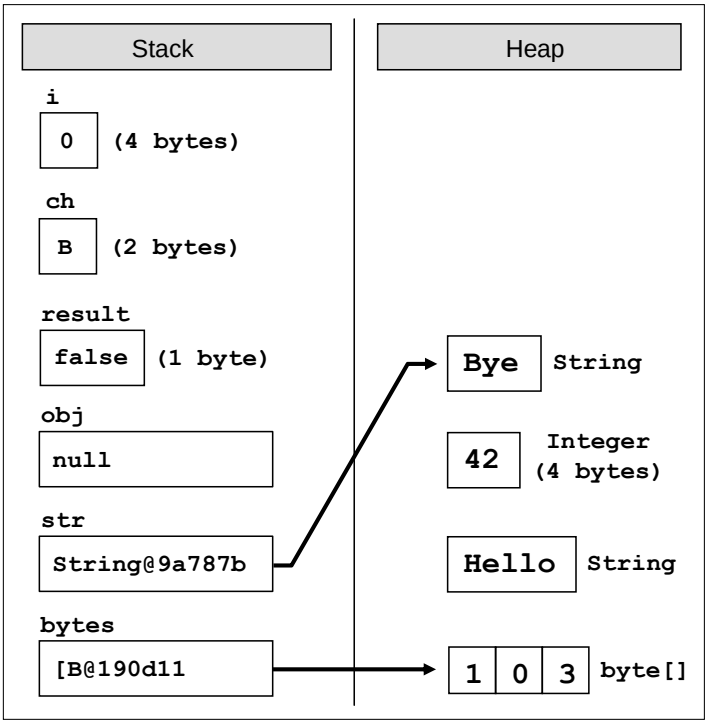


Ако сега изпълним следния код, който променя стойностите на променливите, ще видим какво се случва с паметта при промяна на стойностни и референтни типове:

```
i = 0;
ch = 'B';
result = false;
obj = null;
str = "Bye";
```

```
bytes[1] = 0;
```

След тези промени променливите и техните стойности са разположени в паметта по следния начин:



Както можете да забележите от фигурата, при промяна на стойностен тип (**i=0**) се променя директно стойността му в стека.

При промяна на референтен тип се променя директно стойността му в динамичната памет (**bytes[1]=0**). Променливата, която държи референцията, остава непроменена (**B@190d11**). При записване на стойност **null** в референтен тип съответната референция се разкача от стойността си и променливата остава без стойност (**obj=null**).

При присвояване на нова стойност на обект (референтен тип) новият обект се заделя в динамичната стойност, а старият обект остава свободен. Референцията се пренасочва към новия обект (**str="Bye"**). Старите обекти, понеже не се използват, ще бъдат почистени по някое време от системата за почистване на паметта (garbage collector).

Литерали

Примитивните типове, с които се запознахме вече, са специални типове данни, вградени в езика Java. Техните стойности, зададени в сорс кода на програмата се наричат **литерали**. С един пример ще ни стане по-ясно:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 20000;
int i = 300000;
```

В примера литерали са **true**, **'C'**, **100**, **20000** и **300000**. Те представляват стойности на променливи, зададени непосредствено в сорс кода на програмата.

Видове литерали

Съществуват няколко вида литерали:

- Boolean
- Integer
- Real
- Character
- String
- The **null** literal

Булеви литерали

Булевите литерали са:

- **true**
- **false**

Когато присвояваме стойност на променлива от тип **boolean**, можем да използваме единствено някоя от тези две стойности.

Булеви литерали – пример

Ето пример за декларация на променлива от тип **boolean** и присвояване на стойност, което представлява булевият литерал **true**:

```
boolean result = true;
```

Целочислени литерали

Целочислените литерали представляват поредица от цифри, наставки и представки. Можем да представим целите числа в сорс кода на програмата в десетичен, шестнадесетичен и осмичен формат.

- "**0**" представка означава стойност в осмична бройна система, например **007**;

- "**0x**" и "**0X**" представки означават шестнадесетична стойност, например **0xA8F1**;
- "**l**" и "**L**" наставки означават данни от тип long, например **357L**.

Символът "**l**" лесно се бърка с **1** (едно), затова препоръчително е да използваме символа "**L**".

Целочислени литерали – примери

Ето няколко примера за използване на целочислени литерали:

```
// The following variables are initialized with the same value
int numberInDec = 16;
int numberInOcatal = 020;
int numberInHex = 0x10;

// This will cause an error, because the value 234L is not int
int longInt = 234L;
```

Реални литерали

Реалните литерали, както целочислените, представляват поредица от цифри, знак (**+**, **-**), наставки и символа за десетична запетая. Използваме ги за стойности от тип **float** и **double**. Реалните литерали могат да бъдат представени и в експоненциален формат.

- "**f**" и "**F**" наставки означават данни от тип **float**;
- "**d**" и "**D**" наставки означават данни от тип **double**;
- "**e**" означава експонента, примерно "e-5" означава цялата част да се умножи по 10^{-5}.
- По подразбиране реалните литерали са от тип **double**.

Реални литерали – примери

Ето няколко примера за използване на реални литерали:

```
// The following is the correct way of assigning the value:
float realNumber = 12.5f;

// This is the same value in exponential format:
realNumber = 1.25e+1f;

// The following causes an error because 12.5 is double
float realNumber = 12.5;
```

Символни литерали

Символните литерали представляват единичен символ, ограден в апострофи (единични кавички). Използваме ги за задаване на стойности от тип **char**.

Стойността на този тип литерали може да бъде:

- Символ, примерно **'A'**;
- Код на символ, примерно **'\u0065'**;
- Escaping последователност.

Escaping последователности

Понякога се налага да работим със символи, които не са изписани на клавиатурата или със символи, които имат специално значение, като например символът "нов ред". Те не могат да се изпишат директно в сорс кода на програмата и за да ги ползваме са ни необходими специални техники, които ще разгледаме сега.

Escaping последователностите са литерали, които представят последователност от специални символи, които задават символ, който по някаква причина не може да се изпише директно в сорс кода. Такъв е например символът за нов ред. Те ни дават заобиколен начин (escaping) да напишем някакъв символ.

Примери за символи, които не могат да се изпишат директно в сорс кода, има много: двойната кавичка, табулация, нов ред, наклонена черта и други. Ето някои най-често използваните escaping последователности:

- **\'** – единична кавичка
- **\"** – двойна кавичка
- **** – лява наклонена черта
- **\n** – нов ред
- **\t** – отместване (табулация)
- **\uXXXX** – силвол, зададен с Unicode номера си, примерно **\u03A7**.

Символът **** (лява наклонена черта) се нарича още **екраниращ символ**, защото той позволява да се изпишат на екрана символи, които имат специално значение или действие и не могат да се изпишат в сорс кода.

Escaping последователности – примери

Ето няколко примера за символни литерали:

```
// An ordinary symbol
char symbol = 'a';
System.out.print(symbol);
```

```
// Unicode symbol code in a hexadecimal format
symbol = '\u003A';
System.out.print(symbol);

// Assigning the single quote symbol
symbol = '\'';
System.out.print(symbol);

// Assigning the backslash symbol
symbol = '\\';
System.out.print(symbol);
```

Ако изпълним горния код, ще получим следния резултат:

```
a:'\
```

Литерали за символен низ

Литералите за символен низ се използват за данни от тип **String**. Състоят се от текстова стойност, заградена в двойни кавички и представляват последователност от символи.

За символните низове важат всички правила за escaping, които важат и за литералите от тип **char**.

Литерали за символен низ – примери

Ето няколко примера за използване на литерали от тип символен низ:

```
String quotation = "\"Hello, Jude\", he said.";
System.out.println(quotation);
String path = "C:\\Windows\\Notepad.exe";
System.out.println(path);
```

Ако изпълним горния код, ще получим следния резултат:

```
"Hello, Jude", he said.
C:\Windows\Notepad.exe
```

Повече за символните низове ще намерите в главата "Символни низове".

Упражнения

1. Декларирайте няколко променливи, като изберете за всяка една най-подходящия от типовете **byte**, **short**, **int** и **long**, за да представят следните стойности: 52130; -115; 4825932; 97; -10000.

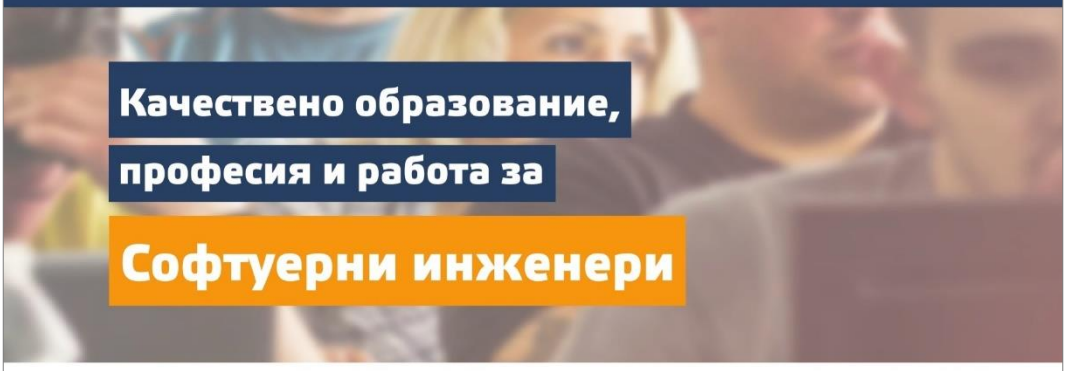
2. Кои от следните стойности може да се присвоят на променлива от тип **`float`** и кои на променлива от тип **`double`**: 34.567839023; 12.345; 8923.1234857; 3456.091?
3. Инициализирайте променлива от тип **`int`** със стойност 256 в шестнадесетичен формат (256 е 100 в бройна система с база 16).
4. Декларирайте променлива **`isMale`** от тип **`boolean`** и присвоете стойност на последната в зависимост от вашия пол.
5. Декларирайте две променливи от тип **`String`** със стойности "Hello" и "World". Декларирайте променлива от тип **`Object`**. Присвоете на тази променлива стойността, която се получава от конкатенацията на двете стрингови променливи (добавете интервал, ако е необходимо). Отпечатайте променливата от тип **`Object`**. Декларирайте променлива от тип **`String`** и присвоете на последната променливата от тип **`Object`**.
6. Декларирайте променлива от тип **`String`**, отпечатайте я на конзолата и получете следното "The "use" of quotations causes difficulties." (без първите и последни кавички).
7. Напишете програма, която принтира на конзолата равнобедрен триъгълник, като страните му са очертани от символа звездичка "*".
8. Напишете програма, която принтира фигура във формата на сърце със знака "o".
9. Фирма, занимаваща се с маркетинг, иска да пази запис с данни на нейните служители. Всеки запис трябва да има следната характеристика – първо име, фамилия, възраст, пол (м или ж), ID номер и уникален номер на служителя (27560000 до 27569999). Декларирайте необходимите променливи, нужни за да се запази информацията за един служител, като използвате подходящи типове данни и описателни имена.
10. Декларирайте две променливи от тип **`int`**. Задайте им стойности съответно 5 и 10. Разменете стойностите им и ги отпечатайте.

Решения и упътвания

1. Погледнете размерността на числените типове.
2. Имайте в предвид броят символи след десетичния знак. Направете справка в таблицата с размерите на типовете **`float`** и **`double`**.
3. Вижте секцията за целочислени литерали.
4. Вижте секцията за булеви променливи.
5. Вижте секциите за символен тип и за обектен тип данни.
6. Погледнете частта за символни литерали. Необходимо е да използвате символа за escaping – наклонена черта "\".
7. Използвайте **`System.out.println()`**.

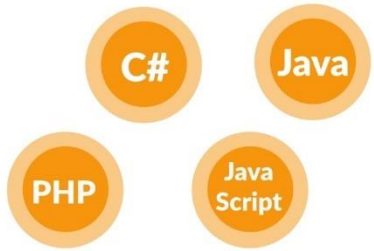
8. Използвайте **`System.out.println()`**.
9. За низовете използвайте тип **`String`**, за пол използвайте тип **`char`** (имаме само един символ **м**/**ж**), а за ID номера използваме целочислен тип **`int`**.
10. Използвайте трета временна променлива.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 3. Оператори и изрази

Автор

Лъчезар Божков

В тази тема...

В настоящата тема ще се запознаем с операторите и действията, които те извършват върху различните типове данни. Ще разясним приоритета на операторите и групите оператори според броя на аргументите, които приемат и това какво действие извършват. Във втората част на темата ще разгледаме преобразуването на типове, защо е нужно и как да се работим с него. Накрая ще разясним какво представляват изразите, след което сме приготвили упражнения, за да ви накараме да разгледате операторите в действие и да затвърдите знанията, който ще придобиете прочитайки главата.

Оператори

Във всички езици за програмиране се използват оператори, чрез които се изразяват някакви действия върху данни. Нека разгледаме операторите в Java и ви покажем за какво служат и как се използват.

Какво е оператор?

След като научихте как да декларирате и назначавате стойности на променливи, вероятно искате да извършите операции с тях. За целта ще се запознаем с операторите. Операторите позволят манипулиране на примитивни типове данни. Те са символи, които извършат специфични операции над един, два или три операнда и връщат резултат от извършените операции. Пример за операторите са символите за събиране, изваждане, делене и умножение в математиката (**+**, **-** , **/**, *****) и операциите, които те извършват върху операндите, над които са приложени.

Операторите в Java

Операторите в Java могат да бъдат разделени в няколко различни категории:

- Аритметични – също както в математиката, служат за извършване на прости математически операции.
- Оператори за присвояване – позволяват присвояването на стойност на променливите.
- Оператори за сравнение – дават възможност за сравнение на два литерала и/или променливи.
- Логически оператори – оператори за работа с логически типове данни.
- Побитови оператори – използват се за извършване на операции върху двоичното представяне на числови данни.
- Оператори за преобразуване на типовете – позволяват преобразуването на данни от един тип в друг.

Категории оператори

Следва списък с операторите, разделени по категории:

Категория	Оператори
аритметични	`-, +, *, /, %, ++, --`
логически	`&&, ||, !, ^`
побитови	`&, |, ^, ~, <<, >>, >>>`
за сравнение	`==, !=, >, <, >=, <=`
за присвояване	`=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`
съединяване на символни низове	`+`
за работа с типове	`(type)`, `instanceof`
други	`., new, (), [], ?:`

Има три основни групи оператори разделени според, това колко аргумента приемат.

Оператори според броя аргументи

Следва списък на групите оператори, според броя аргументите, които приемат:

Тип оператор	Брой на аргументите (операндите)
едноаргументни (unary)	приема един аргумент
двуаргументни (binary)	приема два аргумента
триаргументни (ternary)	приема три аргумента

Всички двуаргументни оператори са ляво-асоциативни, означава, че изразите, в които участват се изчисляват от ляво на дясно, освен операторите за назначаване на стойности. Всички оператори за присвояване на стойности и условният оператор **(:?)** са дясно-асоциативни (изчисляват се от дясно на ляво).

Някой оператори в Java извършват различни операции, когато се приложат с различен тип данни. Пример за това е операторът +. Когато се използва с числени типове данни (**int**, **long**, **float** и др.), операторът извършва операцията математическо събиране. Когато обаче използваме оператора със символни низове, той слепва съдържанието на двете променливи / литерали и връща новополучения низ.

Оператори – пример

Ето един пример за използване на оператори:

```
int z = 4 + 8;
System.out.println(z); // 12

String firstName = "Lachezar";
String lastName = "Bozhkov";

// Do not forget the interval between them
String fullName = firstName + " " + lastName;
System.out.println(fullName); // Lachezar Bozhkov
```

Примерът показва как при използването на **+** с числа операторът връща числова стойност, а при използването му с низове връща низ.

Приоритет на операторите в Java

Някои оператори имат приоритет над други. Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Операторът **()** служи за промяна на приоритета на операторите и се изчислява пръв, също както в математиката.

В таблицата са показани приоритетите на операторите в Java:

Приоритет	Оператори
най-висок	**++, --** (като суфикс), **new, (type)**
	++, -- (като префикс), **+, -** (едноаргументни), **!, ~**
	***, /, %**
	+ (свързване на низове)
	+, -
	<<, >>, >>>

	<, >, <=, >=, instanceof
	==, !=
	&, ^, |
	&&
	||
	?:
най-нисък	**=, *=, /=, %=, +=, -=, <<=, >>=, >>>= &=, ^=, |=**

Операторите, намиращи се по-нагоре в таблицата имат по-висок приоритет от тези, намиращи се след тях, съответно имат предимство при изчисляването на съответния израз. За да се предефинира приоритета може да се използват скоби.

Когато пишем по-сложни или изрази съдържащи повече оператори се препоръчва използването на скоби. Ето един пример:

```
// Ambiguous
x + y / 100

// Unambiguous, recommended
x + (y / 100)
```

Първата операция, която се изпълнява от примера е делението, защото е с приоритет над оператора за събиране. Въпреки това използването на скоби е добра идея, защото кода става по-лесен за чете и възможността да се допусне грешка намалява.

Аритметични оператори

Аритметичните оператори **+, -, *** са същите като в математика. Те извършват събиране, изваждане и умножение върху числови стойности. Когато се използва операторът за деление **/** с целочислен тип (**integer**), върнатият резултат е отново целочислен (без закръгляне). За да се вземе остатъкът от делене на цели числа се използва оператора **%**. Операторът за увеличаване с единица (increment) **++** добавя единица към стойността на променливата, съответно операторът **--** (decrement) изважда единица от стойността.

Когато използваме операторите **++** и **--** като суфикс (поставяме операторът непосредствено пред променливата) първо се пресмята новата стойност, а после се връща резултата и програмата продължава с решението на израза, докато при използването на операторите като постфикс (поставяме оператора непосредствено след променливата) първо се връща оригиналната стойност на операнда, а после се добавя или изважда единица към нея.

Аритметични оператори – примери

Ето няколко примера за аритметични оператори:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
System.out.println(squareSide); // 4.25
System.out.println(squareArea); // 18.0625

int a = 5;
int b = 4;
System.out.println(a + b);      // 9
System.out.println(a + b++);    // 9
System.out.println(a + b);      // 10
System.out.println(a + (++b));  // 11
System.out.println(a + b);      // 11
System.out.println(14 / a);     // 2
System.out.println(14 % a);     // 4
```

Логически оператори

Логическите оператори приемат булеви стойности и връщат булев резултат (**true** или **false**). Основните булеви оператори са И (**&&**), ИЛИ (**||**), изключващо ИЛИ (**^**) и логическо отрицание (**!**).

Следва таблица с логическите оператори в Java и операциите, които те извършват:

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

От таблицата, както и от следващия пример става ясно, че логическото "И" връща истина, само тогава, когато и двете променливи съдържат истина. Логическото "ИЛИ" връща истина, когато поне един от операндите е истина. Операторът за логическо отрицание сменя стойността на аргумента. Например, ако операндът е имала стойност true и приложим оператор за отрицание, новата стойност ще бъде **false**. Операторът за отрицание се слага пред аргумента. Изключващото ИЛИ връща резултат true, когато само един от двата операнда има стойност **true**. Ако двата операнда имат различни стойности изключващото ИЛИ ще върне резултат true, ако имат еднакви стойности ще върне **false**.

Логически оператори – пример

Ето един пример за използване на логически оператори. Резултатът от действието на отделните логически оператори е даден като коментари:

```
boolean a = true;
boolean b = false;
System.out.println(a && b);          // false
System.out.println(a || b);          // true
System.out.println(!b);              // true
System.out.println(b || true);       // true
System.out.println((5>7) ^ (a==b)); // false
```

Закони на Де Морган

Логическите операции се подчиняват на законите на Де Морган от математическата логика:

```
!(a && b) == (!a || !b)
!(a || b) == (!a && !b)
```

Първият закон твърди, че отрицанието на конюнкцията (логическо и) на две съждения е равна на дизюнкцията (логическо или) на техните отрицания.

Вторият закон твърди, че отрицанието на дизюнкцията на две съждения е равна на конюнкцията на техните отрицания.

Оператор за съединяване на низове

Оператора **+** се използва за съединяване на символни низове (**String**). Това, което прави операторът е просто слепя два или повече низа и връща резултата като един нов низ. Ако поне един от аргументите в израза е от тип **String**, и има други операнди, които не са от тип **String**, то те автоматично ще бъдат преобразувана към тип **String**.

Оператор за съединяване на низове – пример

Ето един пример, в който съединяваме няколко символни низа:

```
String first = "Star";
String second = "Craft";
System.out.println(first + second); // StarCraft
String output = first + second + " ";
int number = 2;
System.out.println(output + number);
// StarCraft 2
```

В примера инициализираме две променливи от тип **String** и им задаваме стойности. На третия ред съединяваме двата стринга и подаваме резултата

на метода **println(),** за да го отпечата на конзолата. На следващия ред съединяваме двата низа и добавяме интервал накрая. Върнатия резултат записваме в променлива наречена **output**. На последния ред съединяваме съдържанието на низа **output** с числото 2 (съдържанието на променливата **number**) и подаваме резултата отново за отпечатване. Върнатият резултат ще бъде автоматично преобразуван към тип **String**, защото двете променливи са от различен тип.

Конкатенацията (слепването на два низа) на стрингове е бавна операция и трябва да се използва внимателно. Препоръчва се използването на класовете StringBuilder или StringBuffer при нужда от итеративни (повтарящи се) операции върху символни низове.

В главата "Символни низове" ще обясним в детайли защо при операции над символни низове, изпълнени в цикъл, задължително се използват гореспоменатите класове.

Побитови оператори

Побитов оператор означава оператор, който действа над двоичното представяне на числовите типове. В компютрите всички данни и в частност числовите данни се представят като поредица от нули и единици. За целта се използва **двоичната бройна система**. Например числото **55** в двоична бройна система се представя като **00110111**.

Двоичното представяне на данните е удобно, тъй като нулата и единицата в електрониката могат да се реализират чрез логически схеми, в които нулата се представя като "няма ток" или примерно с напрежение -5V, а единицата се представя като "има ток" или примерно с напрежение +5V.

Ще разгледаме в дълбочина двоичната бройна система в главата "Бройни системи", а за момента можем да си представяме, че числата в компютрите се представят като нули и единици и че побитовите оператори служат за анализиране и промяна на точно тези нули и единици.

Побитовите оператори много приличат на логическите. Всъщност можем да си представим, че логическите и побитовите оператори извършат едно и също нещо, но върху различни типове променливи. Логическите оператори работят над стойностите **true** и **false** (булеви стойности), докато побитовите работят над числови стойности и се прилагат побитово, има се предвид **0** и **1** (битове). Също както при логическите оператори, тук има оператор за побитово "И" (**&**), побитово "ИЛИ" (**|**), побитово отрицание (**~**) и изключващо "ИЛИ" (**^**).

Побитови оператори и тяхното действие

Можем да видим символите на операторите и резултата от тяхната употреба в следната таблица:

x	y	~x	x & y	x | y	x ^ y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Както виждаме побитовите и логическите оператори си приличат много. Разликата в изписването на "И" и "ИЛИ" е че при логическите оператори се пише двойни амперсанд или права черта, а при битовите единични. Побитовият и логическият оператор за изключващо или е един и същ "^". За логическо отрицание се използва "**!**", докато за побитово отрицание се използва "**~**".

Има още два побитови оператора, които ги няма при логическите. Това са побитовото преместване в ляво (**<<**) и побитовото преместване в дясно (**>>**). Използвани над числови стойности те преместват всички битове на стойността, съответно на ляво или надясно. Операторите за преместване се използват по следния начин: от ляво на оператора слагаме променливата (операндът), над която ще извършим операцията, вдясно на оператора поставяме число, символизиращо, с колко знака искаме да отместим битовете. **3 << 2** означава, че искаме да преместим два пъти наляво битовете на числото 3. Числото **3** представено в битове изглежда така: **"0000 0011",** когато го преместим два пъти в ляво неговата битова стойност ще изглежда така: **"0000 1100"**, а на тези битове отговаря числото **12**. Ако се вгледаме в примера можем да забележим, че реално сме умножили числото по **4**. Самото побитово преместване може да се представи като умножение (побитово преместване вляво) или делене (преместване в дясно) на променливата на числото **2**. Това явление е следствие от природата на двоичната бройна система.

Побитови оператори – пример

Ето един пример за работа с побитови оператори. Двоичното представяне на числата и резултатите от различните оператори е дадено в коментари:

```
short a = 3;                // 0000 0011 = 3
short b = 5;                // 0000 0101 = 5

System.out.println( a | b); // 0000 0111 = 7
System.out.println( a & b); // 0000 0001 = 1
System.out.println( a ^ b); // 0000 0110 = 6
System.out.println(~a & b); // 0000 0100 = 4
System.out.println(a << 1); // 0000 0110 = 6
System.out.println(a << 2); // 0000 1100 = 12
System.out.println(a >> 1); // 0000 0001 = 1
```

В примера първо създаваме и инициализираме стойностите на две променливи **a** и **b**. По нататък в примера изкарваме на конзолата, резултатите от побитовите операции над двете променливи. Първата операция, която прилагаме е или. От примера се вижда, че за всички позиции, на които е имало **1** в променливите **a** и **b**, има **1** и в резултата. Втората операция е "И". Резултата от операцията съдържа 1 само в най-десния бит, защото и двете променливи имат **1** само в най-десния си бит. Изключващо "ИЛИ" връща единици само там, където **a** и **b** имат различни стойности на битовете. По-надолу можем да видим и резултатите от логическо отрицание и побитово преместване.

Оператори за сравнение

Операторите за сравнение в Java се използват за сравняване на две или повече операнди. Java поддържа шест оператора за сравнение:

- по-голямо (>)
- по-малко (<)
- по-голямо или равно (>=)
- по-малко или равно (<=)
- оператора за равенство (==)
- различие (!=)

Всички оператори за сравнение са двуаргументни (приемат два операнда), а върнатият от тях резултат е булев (**true** или **false**). Операторите за сравнение имат по-малък приоритет от аритметичните, но са с по-голям от операторите за присвояване на стойност.

Оператори за сравнение – пример

Следва примерна програма, която демонстрира употребата на операторите за сравнение в Java:

```
public class RelationalOperatorsDemo {
   public static void main(String args[]) {
      int x = 10, y = 5;
      System.out.println("x > y : " + (x > y)); // true
      System.out.println("x < y : " + (x < y)); // false
      System.out.println("x >= y : " + (x >= y)); // true
      System.out.println("x <= y : " + (x <= y)); // false
      System.out.println("x == y : " + (x == y)); // false
      System.out.println("x != y : " + (x != y)); // true
   }
}
```

В примерната програма, първо създадохме двете променливи **x** и **y** и им присвоихме стойностите 10 и 5. На следващия ред отпечатваме на конзо-

лата, посредством метода **println()** на **System.out**, резултата от сравняването на двете променливи **x** и **y** посредством оператора >. Върнатият резултат е **true**, защото **x** има по-голяма стойност от **y**. На следващите 5 реда се отпечатва върнатият резултат от използването на останалите 5 оператора за сравнение с променливите **x** и **y**.

Оператори за присвояване

Операторът за присвояване на стойност на променливите е "=" (символът равно). Синтаксисът, който се използва за присвояване на стойности е следният:

```
операнд1 = литерал или операнд2;
```

Оператори за присвояване – пример

Ето един пример, в който използваме операторът за присвояване на стойност:

```
int x = 6;
String helloString = "Здравей стринг.";
int y = x;
```

В горния пример присвояваме стойност 6 на променливата x. На втория ред присвояваме текстов литерал на променливата **helloString**, а на третия ред копираме стойността от променливата **x** в променливата **y**.

Каскадно присвояване

Операторът за присвояване може да се използва и каскадно (да се използва повече от веднъж в един и същ израз). В този случай присвояванията се извършват последователно отдясно наляво. Ето един пример:

```
int x, y, z;
x = y = z = 25;
```

На първия ред от примера създаваме три променливи, а на втория ред ги инициализираме със стойност 25.

Операторът за присвояване в Java е "=", докато операторът за сравнение е "==". Размяната на двата оператора е честа причина за грешки при писането на код. Внимавайте да не объркате оператора за сравнение с оператора за присвояване.

Комбинирани оператори за присвояване

Освен оператора за присвояване в Java има и комбинирани оператори за присвояване. Те спомагат за съкращаването на обема на кода, като

позволяват изписването на две операции чрез един оператор. Комбинираните оператори имат следния синтаксис:

```
операнд1 оператор = операнд2;
```

Горният израз е идентичен със следния:

```
операнда1 = операнда1 оператор операнда2;
```

Ето един пример за комбиниран оператор за присвояване:

```
int x = 2;
int y = 4;

x *= y; // Same as x = x * y;
System.out.println(x); // 8
```

Най-често използваните комбинирани оператори за присвояване са **+=** (добавя стойността на **операнд2** към **операнд1**), **-=** (изважда стойността на операнда в дясно от стойността на тази в ляво). Други съставни оператори за присвояване са ***=**, **/=** и **%=**.

Следващият пример дава добра представа за комбинираните оператори за присвояване и тяхната употреба:

```
int x = 6;
int y = 4;

System.out.println(y *= 2); // 8
int z = y = 3;              // y=3 and z=3

System.out.println(z);      // 3
System.out.println(x |= 1); // 7
System.out.println(x += 3); // 10
System.out.println(x /= 2); // 5
```

В примера първо създаваме променливите **x** и **y** и им присвояваме стойностите 6 и 4. На следващият ред принтираме на конзолата **y**, след като сме присвоили нова стойност с оператора ***=** и литерала **2.** Резултатът от операцията е 8. По нататък в примера прилагаме други съставни оператори за присвояване и изкарваме получения резултат на конзолата.

Условен оператор ?:

Условния оператор ?: използва булевата стойност от един израз за да определи кой от други два израза да бъде пресметнат и върнат като резултат. Операторът работи над 3 операнда. Символът "**?**" се поставя между първия и втория операнд, а "**:**" се поставя между втория и третия операнд. Първият операнд (или израз) трябва да е от булев тип.

Синтаксисът на оператора е следният:

```
операнд1 ? операнд2 : операнд3
```

Ако **операнд1** има стойност **true**, операторът връща резултат **операнд2**. Ако **операнд1** има стойност **false**, операторът връща резултат **операнд3**.

По време на изпълнение се пресмята стойността на първия аргумент. Ако той има стойност **true**, тогава се пресмята втория (среден) аргумент и той се връща като резултат. Обаче, ако пресметнатият резултат от първия аргумент е **false**, то тогава се пресмята третия (последния) аргумент и той се връща като резултат.

Условен оператор ?: – пример

Ето един пример за употребата на оператора "**?:**":

```
int a = 6;
int b = 4;
System.out.println(a > b ? "a>b" : "b<=a"); // a>b
```

Други оператори

Досега разгледахме аритметичните оператори, логическите и побитовите оператори, оператора за конкатенация на символни низове, също и условният оператор **?:**. Освен тях в Java има още няколко оператора:

- Операторът за достъп "**.**" се използва за достъп до член променливите на даден обект.
- Квадратни скоби **[]** се използват за достъп до елементите на масив.
- Скоби **()** се използват за предефиниране приоритета на изпълнение на изразите и операторите.
- Оператора за преобразуване на типове **(type)** се използва за преобразуване на променлива от един съвместим тип в друг.
- Операторът **new** се използва за създаването и инициализирането на нови обекти.
- Операторът **instanceof** се използва за проверка дали даден обект е съвместим с даден тип.

Други оператори – примери

Ето няколко примера за операторите, които разгледахме в тази секция:

```
int a = 6;
int b = 3;
int c = 3;
```

```
System.out.println(c);          // 3
System.out.println((a+b) / 2); // 4

String s = "Beer";
System.out.println(s instanceof String); // true

int d = 0;
System.out.println(d);          // 0
System.out.println((a+b) / d); // ArithmeticException
```

Преобразуване на типовете

Операторите работят върху еднакъв тип данни. Въпреки това в Java има голямо разнообразие от типове данни, от които можем да избираме най-подходящия за определената цел. За да извършим операция върху променливи от два различни типа данни ни се налага да преобразуваме двата типа към един и същ.

Всички изрази в езика Java имат тип. Този тип може да бъде изведен от структурата на израза и типовете, променливите и литералите използвани в израза. Възможно е да се напише израз, който е с неподходящ тип за конкретния контекст. В някой случаи това ще доведе до грешка в компилацията на програмата, но в други контекста може да приеме тип, който е сходен или свързан с типа на израза. В този случай програмата извършва скрито преобразуване на типове.

Специфично преобразуване от тип **S** към тип **T** позволя на израза от тип **S** да се третира като израз от тип **T** по време на изпълнението на програмата. В някои случай това ще изисква проверка на валидността на преобразуването. Ето няколко примера:

- Преобразуване от тип **Object** към тип **String** ще изисква проверка по време на изпълнение, за да потвърди, че стойността е наистина инстанция от тип **String** или от някои от класовете наследници на **String**.
- Преобразуване от тип **String** към **Object** не изисква проверка. **String** е наследник на **Object** и може да бъде преобразуван към базовия си клас без опасност от грешка или загуба на данни. На наследяването ще се спрем в детайли в главата "Принципи на обектно-ориентираното програмиране".
- Преобразуване от тип **int** към **long** може да се извърши без проверка по време на изпълнението, защото няма опасност от загуба на данни.
- Преобразуване от тип **double** към **long** изисква преобразуване от 64-битова плаваща стойност към 64-битова целочислена. В зависимост от стойността, може да се получи загуба на данни, заради това е необходимо изрично преобразуване на типа.

В Java не всички типове могат да бъдат преобразувани във всички други, а само към някои определени. За удобство ще групираме някой от възможните преобразувания в Java според вида им в две категории:

- Скрито преобразуване;
- Изрично преобразуване.

Неявно (implicit) преобразуване на типове

Неявното (скритото) преобразуване на типове е възможно единствено, когато няма възможност от загуба на данни при преобразуването, тоест когато конвертираме от тип с по-малък обхват към тип с по-голям (примерно от int към **long**). За да направим неявно преобразуване не е нужно да използваме какъвто и да е оператор, затова се нарича скрито. Преобразуването става автоматично от компилатора, когато присвояваме стойност от по-малък обхват в променлива с по-голям обхват или когато в израза има типове с различен обхват. Тогава преобразуването става към типа с по-голям обхват.

Неявно преобразуване на типове – пример

Ето един пример за неявно (implicit) преобразуване на типове:

```
int myInt = 5;
System.out.println(myInt); // 5

long myLong = myInt;
System.out.println(myLong); // 5

System.out.println(myLong + myInt); // 10
```

В примера създаваме променлива **myInt** от тип **int** и присвояваме стойност **5**. По-надолу създаваме променлива **myLong** от тип **long** и задаваме стойността, съдържаща се в **myInt**. Стойността запазена в **myLong**, автоматично се конвертира от тип **int** към тип **long**. Накрая в примера изкарваме резултата от събирането на двете променливи. Понеже променливите са от различен тип, те автоматично се преобразуват към типа с по-голям обхват, тоест към **long** и върнатият резултат, който се отпечатва на конзолата, отново е **long**. Всъщност подадения параметър на метода **println()** е от тип **long**, но вътре в метода той отново ще бъде конвертиран, този път към тим **String**, за да може да бъде отпечатан на конзолата.

Възможни неявни преобразования

Това са възможните неявни преобразувания на примитивни типове в Java:

- **byte** към **short, int, long, float,** или **double**
- **short** към **int, long, float,** или **double**
- **char** към **int, long, float,** или **double**

- **int** към **long, float,** или **double**
- **long** към **float** или **double**
- **float** към **double**

При преобразуването на типове от по-малък обхват към по-голям няма загуба на данни. Числовата стойност остава същата след преобразуването. Както във всяко правило и тук има малко изключение. Когато преобразуваме тип **int** към тип **float** (32-битови стойности), разликата е, че **int** използва всичките си битове за представяне на едно целочислено число, докато **float** използва част от битовете си за представянето на плаващата запетая. Оттук следва, че е възможно при преобразуване от **int** към **float** да има загуба на точност, поради закръгляне. Същото се отнася при преобразуването на 64-битовите **long** към **double**.

Изрично (explicit) преобразуване на типове

Изричното преобразуване на типове е нужно, когато има вероятност за загуба на данни. Когато конвертираме тип с плаваща запетая към цело-числен тип, винаги има загуба на данни, идваща от плаващата запетая и е задължително използването на изрично преобразуване (**double** към **long**). За да направим такова конвертиране е нужно изрично да използваме оператора за преобразуване на данни (cast оператора): (**type**). Възможно е да има загуба на данни също, когато конвертираме от тип с по-голям обхват към тип с по-малък (**double** към **float** или **long** към **int**).

Изрично преобразуване на типове – пример

Следният пример илюстрира употребата на изрично конвертиране на типовете и загуба на данни:

```
double myDouble = 5.1d;
System.out.println(myDouble); // 5.1

long myLong = (long)myDouble;
System.out.println(myLong); // 5

myDouble = 5e9d; // 5 * 10^9
System.out.println(myDouble); // 5.0E9

int myInt = (int) myDouble;
System.out.println(myInt); // 2147483647
System.out.println(Integer.MAX_VALUE); // 2147483647
```

На първия ред от примера присвояваме стойността 5,1 на променливата **myDouble**. След като я преобразуваме (изрично), посредством оператора **(long)** към тип **long** и изкараме на конзолата променливата **myLong**, виждаме, че променливата е изгубила стойността след плаващата запетая (защото **long** е целочислен тип). След това на седмия ред присвояваме на променливата **myDouble** стойност 5 милиарда. Накрая конвертираме **myDouble**

към **int** посредством оператора **(int)** и разпечатваме променливата **myInt**. Резултатът е същия, както и когато отпечатаме **Integer. MAX_VALUE,** това е така, защото **myDouble** съдържа в себе си по-голяма стойност от обхвата на **int**.

Не винаги е възможно да се предвиди каква ще бъде стойността на дадена променлива след препълване на обхвата и! Затова използвайте достатъчно големи типове и внимавайте при преминаване към "по-малък" тип.

Загуба на данни при преобразуване на типовете

Пример за загуба на информация при преобразуване на типове:

```
long myLong = Long.MAX_VALUE;
int myInt = (int)myLong;

System.out.println(myLong); // 9223372036854775807
System.out.println(myInt); // -1
```

Операторът за преобразуване може да се използва и при неявно преобразуване по-желание. Това допринася за четливостта на кода, намалява шанса за грешки и се счита за добра практика от много програмисти.

Ето още няколко примера за преобразуване на типове:

```
float heightInMeters = 1.74f;    // Explicit conversion
double maxHeight = heightInMeters;          // Implicit
double minHeight = (double) heightInMeters; // Explicit
float actualHeight = (float) maxHeight;     // Explicit

float maxHeightFloat = maxHeight; // Compilation error!
```

В примера на последния ред имаме израз, който ще генерира грешка при компилирането. Това е така, защото се опитваме да конвертираме неявно от тип **double** към тип **float**, от което може да има загуба на данни. Java е строго типизиран език за програмиране и не позволява такъв вид присвояване на стойности.

Възможни изрични преобразования

Това са възможните явни (изрични) преобразувания и при всички тях има възможност за загуба на данни, така че внимавайте:

- **short** към **byte** или **char**
- **char** към **byte** или **short**
- **int** към **byte**, **short** или **char**
- **long** към **byte**, **short**, **char** или **int**
- **float** към **byte**, **short**, **char**, **int** или **long**

- **double** към **byte**, **short**, **char**, **int**, **long** или **float**

Тези преобразувания могат да изгубят, както информация за големината на числото, така и информация за неговата точност (precision).

Когато преобразуваме **byte** към char имаме първо скрито конвертиране от byte към **int**, а след него изрично преобразуване от **int** към **char**.

Преобразуване към символен низ

При необходимост можем да преобразуваме към низ, всеки отделен тип, включително и стойността **null**. Преобразуването на символни низове става автоматично винаги, когато използваме оператора за конкатенация и някой от аргументите не е от тип низ. В този случай аргумента се преобразува към низ и операторът връща нов низ представляващ конкатенацията на двата низа.

Друг начин да преобразуваме различни обекти към тип низ е като извикаме метода **toString()** на променливата.

Преобразуване към символен низ – пример

Нека разгледаме няколко примера за преобразуване на различни типове данни към символен низ:

```
int a = 5;
int b = 7;
String s = "Sum=" + (a + b);
System.out.println(s);

String incorrect = "Sum=" + a + b;
System.out.println(incorrect);

System.out.println("Perimeter = " + 2 * (a + b) +
   ". Area = " + (a * b) + ".");
```

Резултатът от изпълнението на примера е следният:

```
Sum=12
Sum=57
Perimeter = 24. Area = 35.
```

От резултата се вижда, че долепването на число към символен низ връща като резултата символния низ, следван от текстовото представяне на числото. Забележете, че операторът "**+**" за залепване на низове може да предизвика неприятен ефект при събиране на числа, защото има еднакъв приоритет с оператора "**+**" за събиране. Освен, ако изрично не променим приоритета на операциите чрез поставяне на скоби, те винаги се изпълняват отляво надясно.

Изрази

Голяма част от работата на една програма е пресмятане на изрази. Изразите представляват поредици от оператори, литерали и променливи, които се изчисляват до определена стойност от някакъв тип (число, стринг, обект или друг тип). Ето няколко примера за изрази:

```
int r = (150-20) / 2 + 5;

// Expression for calculation of the surface of the circle
double surface = Math.PI * r * r;

// Expression for calculation of the perimeter of the circle
double perimeter = 2 * Math.PI * r;

System.out.println(r);
System.out.println(surface);
System.out.println(perimeter);
```

В примера са дефинирани три израза. Първият израз пресмята радиуса на дадена окръжност. Вторият пресмята площта на окръжността, а последният намира периметърът ѝ. Ето какъв е резултатът е изпълнения на горния програмен фрагмент:

```
70
15393.804002589986
439.822971502571
```

Изчисляването на израз може да има и странични действия, защото изразът може да съдържа вградени оператори за присвояване, увеличаване или намаляване на стойност (increment, decrement) и извикване на методи. Ето пример за такъв страничен ефект:

```
int a = 5;
int b = ++a;

System.out.println(a); // 6
System.out.println(b); // 6
```

Упражнения

1. Напишете израз, който да проверява дали дадено цяло число е четно или нечетно.
2. Напишете булев израз, който да проверява дали дадено цяло число се дели на 5 и на 7 без остатък.
3. Напишете израз, който да проверява дали дадено цяло число съдържа 7 за трета цифра (отдясно на ляво).

4. Напишете израз, който да проверява дали третия бит на дадено число е 1 или 0.
5. Напишете програма, която за подадени дължина и височина на правоъгълник, изкарват на конзолата неговият периметър и лице.
6. Напишете израз, който изчислява площта на трапец по дадени a, b и h.
7. Силата на гравитационното поле на луната е приблизително 17% от това на земята. Напишете програма, която да изчислява тежестта на човек на луната, по дадената тежест на земята.
8. Напишете програма, която проверява дали дадена точка O (x, y) е вътре в окръжността K ((0,0), 5).
9. Напишете програма, която проверява дали дадена точка O (x, y) е вътре в окръжността K ((0,0), 5) и е извън правоъгълника ((-1, 1), (5, 5).
10. Напишете програма, която приема за вход четирицифрено число във формат *abcd* и след това извършва следните действия върху него:
    - Пресмята сбора от цифрите на числото.
    - Разпечатва на конзолата цифрите в обратен ред: *dcba*.
    - Поставя последната цифра, на първо място: *dabc*.
    - Разменя мястото на втората и третата цифра: *acbd*.
11. Дадено е число n и позиция p. Напишете поредица от операции, които да отпечатат стойността на бита на позиция p от числото n (0 или 1). Пример: n=35, p=5 -> 1. Още един пример: n=35, p=6 -> 0.
12. Дадено е число n, стойност v (v = 0 или 1) и позиция p. Напишете поредица от операции, които да променят стойността на n, така че битът на позиция p да има стойност v. Пример n=35, p=5, v=0 -> n=3. Още един пример: n=35, p=2, v=1 -> n=39.
13. Напишете програма, която проверява дали дадено число n (n < 100) е просто.

Решения и упътвания

1. Вземете остатъкът от деленето на числото на 2 и проверете дали е 0 или 1 (четно, нечетно).
2. Ползвайте логическо "И".
3. Разделете числото на 100 и го запишете в нова променлива. Нея разделете на 10 и вземете остатъкът. Остатъкът от делението на 10 е третата цифра от първоначалното число. Проверете равна ли е на 7.
4. Използвайте побитово "И" върху числото и число, което има 1 само на третия бит. Ако върнатият резултат е различен от 0, то третия бит е 1.
5. Използвайте класа за четене от конзолата.

6. Формула за лице на трапец: **S = (a + b) / 2 * h**.
7. Използвайте следния код, за да прочетете число от конзолата:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter number:");
int number = input.nextInt();
```

8. Използвайте питагоровата теорема **c2 = a2 + b2**. За да е вътре в кръга, то **c** следва да е по-малко от **5**.
9. Използвайте кода от задача 8 и добавете проверка за правоъгълника.
10. За да вземете отделните цифри на числото, можете да го делите на 10 и да взимате остатъка 4 последователни пъти.
11. Ползвайте побитови операции:

```
int n = 35; // 00100011
int p = 6;
int i = 1; // 00000001
int mask = i << p; // Move the 1st bit with p positions

// If i & mask are positive then the p-th bit of n is 1
System.out.println((n & mask) != 0 ? 1 : 0);
```

12. Ползвайте побитови операции, по аналогия с предната задача.
13. Прочетете за цикли в Интернет. Използвайте цикъл и проверете числото за делимост на всички числа от 1 до корен квадратен от числото.

Глава 4. Вход и изход от конзолата

Автор

Борис Вълков

В тази тема...

В настоящата тема ще се запознаем с конзолата. Ще обясним какво представлява тя, кога и как се използва, какви са принципите на повечето програмни езици за достъп до конзолата. Ще се запознаем с някои от възможностите на Java за взаимодействие с потребителя. Ще разгледаме основните потоци за входно-изходни операции `System.in`, `System.out` и `System.err`, класът `Scanner` и използването на форматиращи низове за отпечатване на данни в различни формати.

Какво представлява конзолата?

Конзолата представлява прозорец на операционната система, през който потребителите могат да си взаимодействат с програмите от ядрото на операционната система или с другите конзолни приложения. Взаимодействието става чрез въвеждане на текст от стандартния вход (най-често клавиатурата) или извеждане на текст на стандартния изход (най-често на екрана на компютъра). Тези операции са известни още, като входно-изходни. Текстът, изписван на конзолата носи определена информация и представлява поредица от символи изпратени от една или няколко програми.

За всяко конзолно приложение операционната система свързва устройства за вход и изход. По подразбиране това са клавиатурата и екрана, но те могат да бъдат пренасочвани към файл или други устройства.

Комуникация между потребителя и програмата

Голяма част от програмите си комуникират по някакъв начин с потребителя. Това е необходимо, за да може потребителя да даде своите инструкции към системата. Съвременните начини за комуникация са много и различни, те могат да бъдат през графичен или уеб-базиран интерфейс, конзола или други. Както споменахме, едно от средствата за комуникация между програмите и потребителят е конзолата, но тя става все по-рядко

използвана. Това е така, понеже съвременните средства за комуникация са по-удобни и интуитивни за работа.

Кога да използваме конзолата?

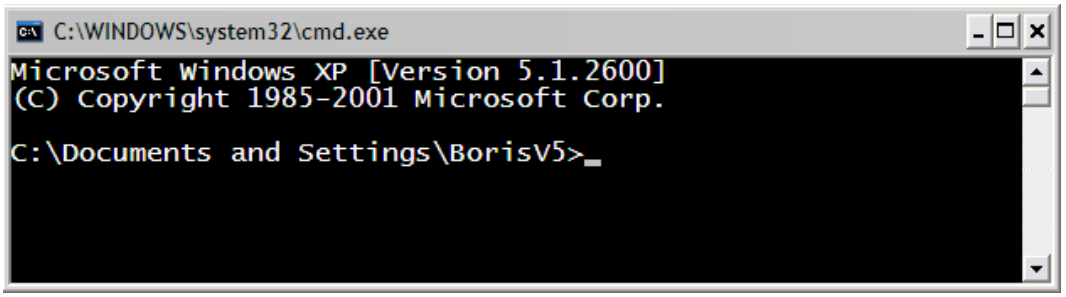
В някои случаи, конзолата си остава незаменимо средство за комуникация. Един от тези случаи е писане на малки и прости програмки, където по-важното е вниманието да е насочено към конкретния проблем, който решаваме, а не към елегантно представяне на резултата на потребителя. Тогава се използва просто решение за въвеждане или извеждане на резултат, каквото е конзолният вход-изход.

Как да стартираме конзолата?

Всяка операционна система си има собствен начин за стартиране на конзолата. Под Windows например стартирането става по следния начин:

```
Start -> (All) Programs -> Accessories -> Command Prompt
```

След стартиране на конзолата, трябва да се появи черен прозорец, който изглежда по следния начин:



При стартиране на конзолата, за текуща директория се използва личната директория на текущия потребител, която се извежда като ориентир за потребителя.

Подробно за конзолите

Конзолата още наричана "**Command Prompt**" или "**shell**" или "команден интерпретатор" е програма на операционната система, която осигурява достъп до системни команди, както и до голям набор от програми, които са част от операционната система или са допълнително инсталирани.

Думата "**shell**" (шел) означава "обвивка" и носи смисъла на обвивка между потребителя и вътрешността на операционната система (ядрото).

Така наречените "обвивки", могат да се разгледат в две основни категории, според това какъв интерфейс могат да предоставят към операционната система.

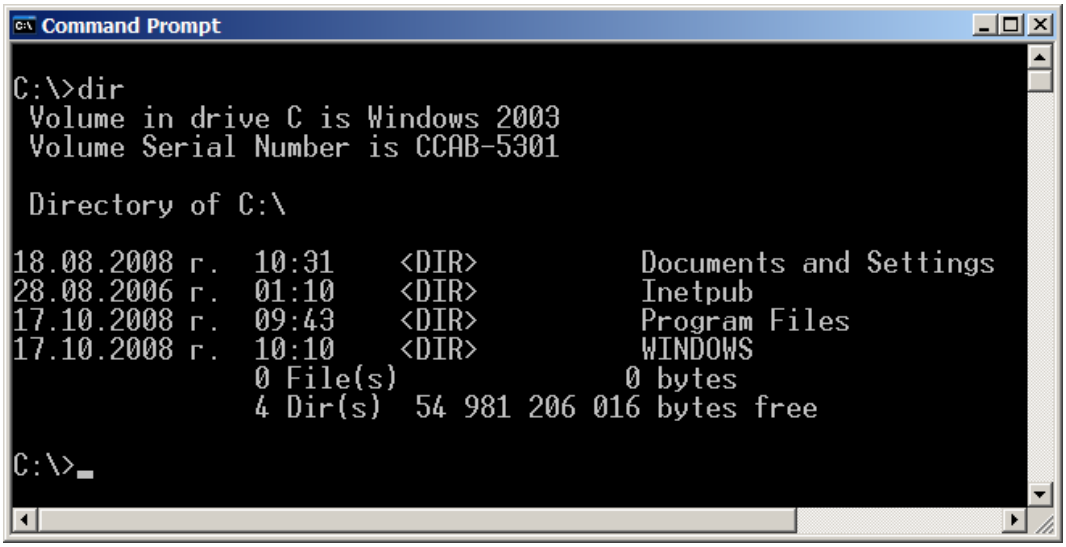
- Команден интерфейс (CLI – Command Line Interface) – представлява конзола за команди (като например **cmd.exe**).

- Графичен интерфейс (GUI – Graphical User Interface) – представлява графична среда за работа (като например Windows Explorer).

И при двата вида, основната цел на обвивката е да стартира други програми, с които потребителят работи, макар че повечето интерпретатори поддържат и разширени функционалности, като например възможност за разглеждане съдържанието на директориите.

Всяка операционна система има свой команден интерпретатор, който дефинира собствени команди.

Например при стартиране на конзолата на Windows в нея се изпълнява т. нар. команден интерпретатор на Windows (**cmd.exe**), който изпълнява системни програми и команди в интерактивен режим. Например командата **dir**, показва файловете в текущата директория:



Основни конзолни команди

Ще разгледаме някои базови конзолни команди, които ще са ни от полза при намиране и стартиране на програми.

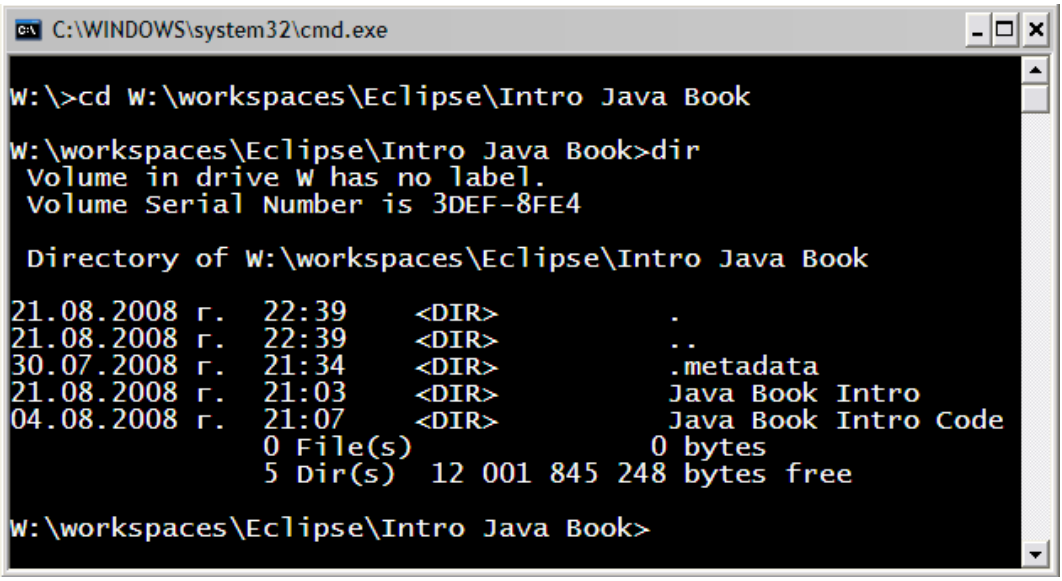
Конзолни команди под Windows

Командният интерпретатор (конзолата) се нарича "**Command Prompt**" или "**MS-DOS Prompt**" (в по-старите версии на Windows). Ще разгледаме няколко базови команди за този интерпретатор:

Команда	Описание
dir	Показва съдържанието на текущата директория.

`cd <directory name>`	Променя текущата директория.
`mkdir <directory name>`	Създава нова директория в текущата.
`rmdir <directory name>`	Изтрива съществуваща директория.
`type <file name>`	Визуализираща съдържанието на файл.
`copy <src file> <destination file>`	Копира един файл в друг файл.

Ето пример за изпълнение на няколко команди в командния интерпретатор на Windows. Резултатът от изпълнението на командите се визуализира в конзолата:



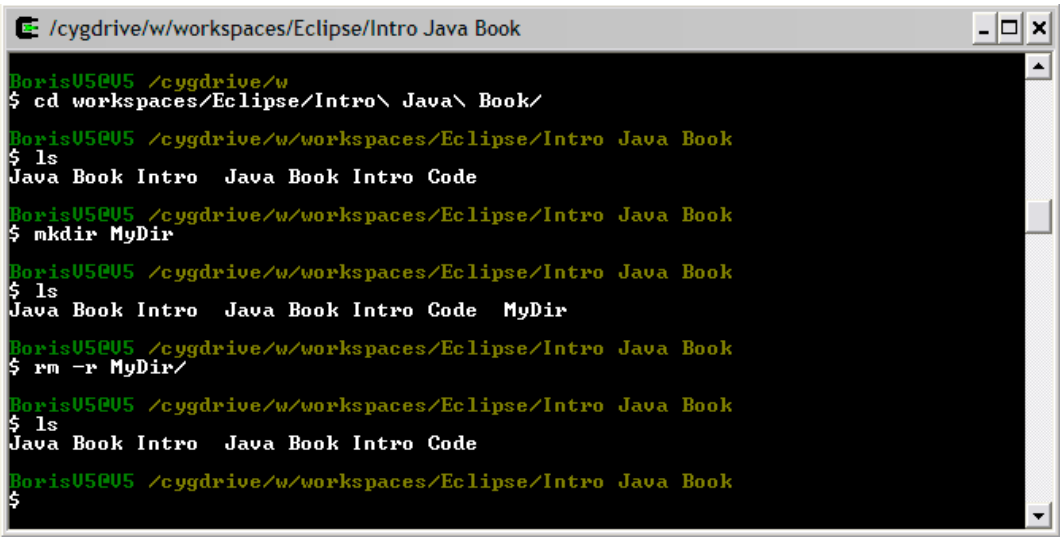
Конзолни команди под Linux

Командният интерпретатор в Linux се различава доста от този на Windows, но основните команди са подобни:

Команда	Описание
`cd <directory name>`	Променя текущата директория.
`ls`	Показва съдържанието на текущата директория.
`mkdir <directory name>`	Създава нова директория в текущата.
`rm -r <directory name>`	Изтрива съществуваща директория заедно с всички файлове и поддиректории в нея рекурсивно.
`cat <file name>`	Показва съдържанието на файл.

`cp <src file> <destination file>`	Копира един файл в друг файл.

Ето пример за изпълнение на няколко команди в командния интерпретатор на Linux. Ето пример за изпълнение на няколко Linux команди:



В примера е използван **Cygwin**. Той представлява колекция от инструменти, които се инсталират като разширение на Windows и могат да изпълняват Linux софтуер в Windows среда.

Cygwin може също да изпълнява системни Linux команди в идващия с него команден интерпретатор "Cygwin Bash Shell". При този интерпретатор, командата "**cd**" има една особеност. Налага се да се ползва обратна наклонена черта, ако името на директорията съдържа интервали.

Стандартен вход-изход

Стандартният вход-изход известен още, като "**Standard I/O**" е системен входно-изходен механизъм създаден още от времето на Unix операционните системи. За вход и изход се използват специални периферни устройства, чрез които може за се въвеждат и извеждат данни.

Когато програмата е в режим на приемане на информация и очаква действие от страна на потребителя, в конзолата започва да мига курсор, подсказващ за очакванията на системата.

По-нататък ще видим как можем да пишем Java програми, които очакват въвеждане на входни данни от конзолата.

Печатане на конзолата

В повечето програмни езици отпечатване и четене на информация в конзолата е реализирано по различен начин, но повечето решения се базират на концепцията за "стандартен вход" и "стандартен изход".

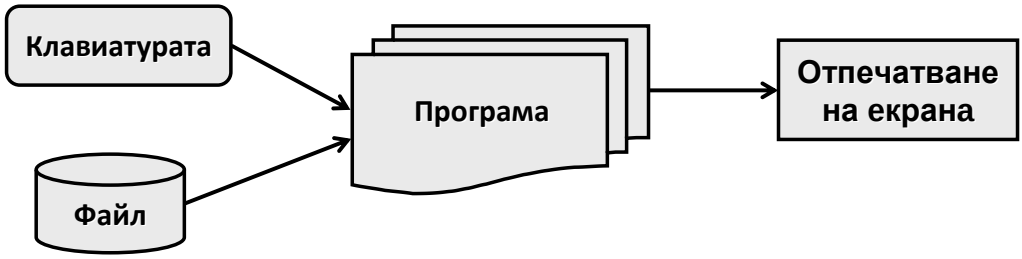
Стандартен вход и стандартен изход

Операционната система е длъжна да дефинира стандартни входно-изходни механизми за взаимодействие с потребителя. При стартиране на дадена програма, служебен код изпълняван преди тази програма е отговорен за отварянето (затварянето) на потоци, към предоставените от операционната система механизми за вход-изход. Този служебен код инициализира програмната абстракция, за взаимодействие с потребителя, заложена в съответния език за програмиране. По този начин стартираното приложение може да чете наготово потребителски вход от стандартния входен поток (в Java това е **`System.in`**), може да записва информация в стандартния изходен поток (в Java това е **`System.out`**) и може да съобщава проблемни ситуации в стандартния поток за грешки (в Java това е **`System.err`**).

Концепцията за потоците ще бъде подробно разгледана по-късно. Засега ще се съсредоточим върху теоретичната основа, засягаща програмния вход и изход в Java. За повече информация по темата вижте секцията "Четене от потока System.in".

Устройства за конзолен вход и изход

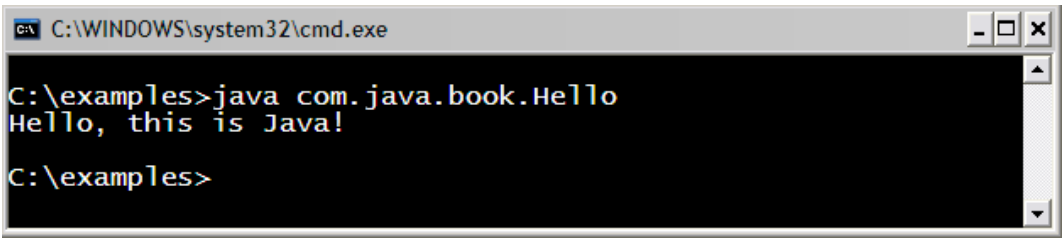
Освен от клавиатура, входът в едно приложение може да дойде от много други места, като например файл, микрофон, бар-код четец и др. Изходът от една програма може да е на конзолата (на екрана), както и във файл или друго изходно устройство, например принтер:



Ще покажем базов пример онагледяващ отпечатването на текст в конзолата чрез абстракцията за достъп до стандартния вход и стандартния изход, предоставена ни от Java:

```
System.out.println("Hello, this is Java!");
```

Резултатът от изпълнението на горния код би могъл да е следният:



Потокът System.out

Класът **java.lang.System** има различни методи (класовете се разглеждат подробно в главата "Създаване и използване на обекти"), повечето са свързани със системни функционалности (взимане на свойства от операционната система, системно време и т.н.). Този клас е част от стандартните пакети на Java. Това което прави впечатление, е че класът има три публични, статични променливи (**in**, **out** и **err**). Причината тези полета да са публични и статични е да може да се използват директно, без да има нужда да се създава инстанция на класа **System**.

Всяко от полетата за стандартен вход-изход е от определен тип (клас), който ни определя позволените операции, които могат да бъдат извършвани. Най-използваните действия, са тези за четене и писане. Обектите **System.out** и **System.err** са от тип **PrintStream** и чрез тези обекти се извършват предимно операции за писане, докато обекта **System.in** е от тип **InputStream** и извършва предимно операции за четене.

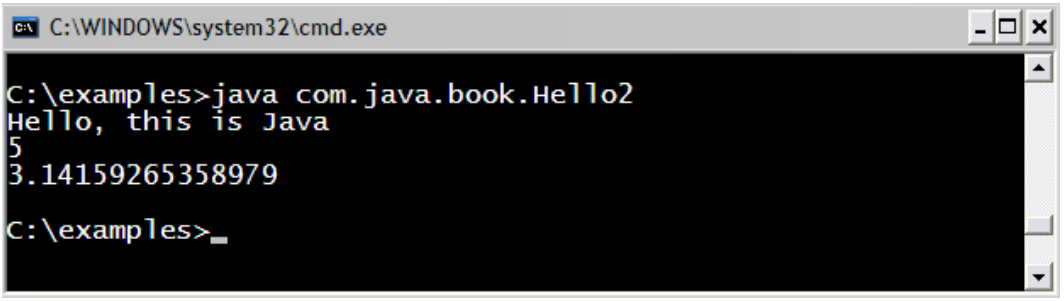
Използване на print() и println()

Работата със съответните методи е безпроблемна, понеже може да се отпечатват всички основни типове (стринг, числени и примитивни типове):

Ето някой примери за отпечатването на различни типове данни:

```
// Print String
System.out.println("Hello, this is Java");

// Print int
System.out.println(5);

// Print double
System.out.println(3.14159265358979);
```

Резултатът от изпълнението на този код изглежда така:



Както виждаме, чрез **System.out.println** е възможно да отпечатаме различни типове, това е така понеже **за всеки от типовете има предефинирана версия на метода println() в класа PrintStream** (може да се уверим сами като погледнем класа **PrintStream** в API-то на Java).

Разликата между print(…) и println(…), е че метода **print(…)** отпечатва в конзолата това, което му е подадено между скобите, но не прави нищо допълнително. Докато метода **println(…)** е съкращение на "print line", което означава "отпечатай линия". Този метод прави това, което прави **print(…)**, но в допълнение отпечатва и нов ред. В действителност методът не отпечатва нов ред, а просто слага "команда" за преместване на курсора на позицията, където започва новият ред.

Ето един пример, който илюстрира разликата между **print** и **println**:

```
System.out.println("I love");
System.out.print("this ");
System.out.print("Book!");
```

Изходът от този пример е:

```
I love
this Book!
```

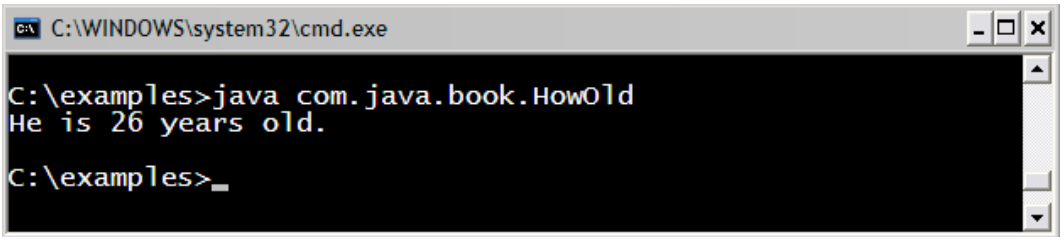
Забелязваме, че изхода от примера е отпечатан на два реда, независимо че кодът е на три. Това е така, понеже на първия ред от кода използваме **println(),** по този начин се отпечатва "**I love**" и след това се минава на нов ред. В следващите два реда от кода се използва метода **print,** които печата, без да минава на нов ред и по този начин думите "**this**" и "**Book!**" си остават на един ред.

Конкатенация на стрингове

В общия случай Java не позволява използването на оператори върху стрингови обекти. Единственото изключение на това правило е операторът за събиране (+), който конкатенира (събира) два стринга, връщайки като резултат нов стринг. Това позволява навързването на верига от конкатениращи (+) операции. Следващия пример показва конкатенация на три стринга.

```
String age = "26"; // String, not number
String text = "He is " + age + " years old.";
System.out.println(text);
```

Резултатът от изпълнението на този код е отново стринг:



Конкатенация на смесени типове

Какво се случва, когато искаме да отпечатаме по-голям и по-сложен текст, който се състои от различни типове? До сега използвахме версиите на метода **println** за точно определен тип. Нужно ли е, когато искаме да отпечатаме различни типове наведнъж, да използваме различните версии на метода **print** за всеки един от тези типове? Не, това не е нужно! Решението на този въпрос е в следващия пример. Разликата на този пример с горния, е, че годините (**age**) са от целочислен тип, който е **различен** от стринг:

```
int age = 26; // int, no String
String text = "He is " + age + " years old.";
System.out.println(text);
```

В примера се извършва конкатенация и отпечатване. Резултатът от примера е същият:

```
He is 26 years old.
```

На втори ред от кода на примера виждаме, че се извършва операцията събиране (конкатенация) на стринга "**He is**" и целочисления тип "**age**". Опитваме се да съберем два различни типа. Това е възможно поради наличието на следващото важно правило.

Когато стринг участва в конкатенация с какъвто и да е друг тип, резултатът винаги е стринг.

От правилото става ясно, че резултата от "**He is**" + **age** е отново стринг, след което резултатът се събира с последната част от израза "**years old.**". Така след извикване на верига от оператори за събиране, в крайна сметка се получава един стринг и съответно се извиква стринговата версия на метода **println(…)**.

За краткост, горния пример може да бъде написан по следния начин:

```
int age = 26;
System.out.println("He is " + age + " years old.");
```

Особености при конкатенация на низове

Има ситуации при конкатенацията (съединяването) на низове, за които трябва да знаем и да внимаваме. Следващият пример показва изненадващо поведение на код, който на пръв поглед изглежда нормално:

```
String s = "Four: " + 2 + 2;
System.out.println(s); // Four: 22

String s = "Four: " + (2 + 2);
System.out.println(s); // Four: 4
```

Редът на изпълнение на операторите (вж. главата "Оператори и изрази") е от голямо значение! В примера първо се извършва събиране на "**Four:** " с "**2**" и **резултатът от операцията е стринг**. Следва повторна конкатенация с второто число, от където се получава неочакваното слепване на резултата "**Four: 22**" вместо очакваното "**Four: 4**". Това е така, понеже операциите се изпълняват от ляво на дясно и винаги участва стринг в конкатенацията.

За да се избегне тази неприятна ситуация може да се използват скоби, които ще променят реда на изпълнение на операторите и ще се постигне желания резултат. Скобите, като оператори с най-голям приоритет, принуждават извършването на операцията "събиране" на двете числа да настъпи преди конкатенацията със стринг, така коректно се извършва първо събиране на двете числа.

Посочената грешка е често срещана при начинаещи програмисти, защото те не съобразяват, че конкатенирането на символни низове се извършва отляво надясно, защото събирането на числа не е с по-висок приоритет.

Когато конкатенирате низове и събирате числа, използвайте скоби, за да укажете правилния ред на операциите. Иначе те се изпълняват отляво надясно.

Форматиран изход с printf()

За отпечатването на дълги и сложни поредици от елементи е въведен специален метод наречен **printf(…)**. Този метод е съкращение на "**Print F**ormatted". Метода е известен и широко използван в средите на "C", макар и да не води началото си от този език за програмиране (а от BCPL).

Методът **printf(…)** има съвсем различна концепция от тази на стандартните методи за печатане в Java. Основната идея на **printf(…)** е да приеме специален стринг, форматиран със специални форматиращи символи и списък със стойностите, които трябва да се заместят на мястото на

"форматните спецификатори". Ето как е дефиниран **printf(…)** в стандартните библиотеки на Java:

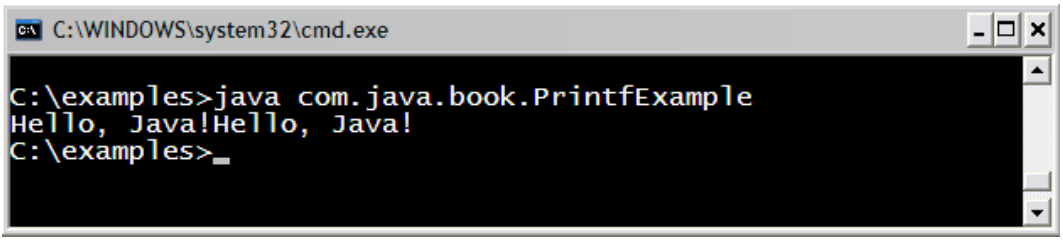
```
printf(<formatted string>, <param1>, <param2>, <param3>, …)
```

Форматиран изход с printf() – примери

Следващият пример отпечатва два пъти едно и също нещо, но по различен начин:

```
String str = "Hello, Java!";

// Print normal
System.out.print(str);

// Print formatted style
System.out.printf("%s", str);
```

Резултатът от изпълнението на този пример е:



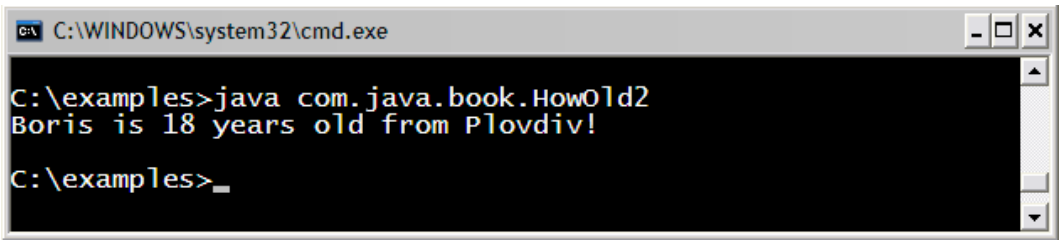
Виждаме като резултат, два пъти "**Hello, Java!**" на един ред. Това е така, понеже никъде в програмата нямаме команда за нов ред.

Първо отпечатваме по познатия ни начин, за да видим разликата с другия подход. Второто отпечатване е форматиращото **printf(…)**. Първия аргумент на метода **printf(…)** е форматиращият стринг. В случая **%s** означава, да се постави **str**, на мястото на **%s**. Методът **printf(…)** е метод на променливата **System.out**, т. е. метод на класа **PrintStream**.

Следващият пример ще разясни допълнително концепцията:

```
String name = "Boris";
int age = 18;
String town = "Plovdiv";
System.out.printf(
  "%s is %d years old from %s!\n", name, age, town);
```

Резултатът от изпълнението на примера е:



От сигнатурата на **printf(…)** видяхме че, първият аргумент е форматиращият низ. Следва поредица от аргументи, които се заместват на местата, където има процент, следван от буква (**%s** или **%d**). Първият **%s** означава да се постави на негово място първия от аргументите, подаден след форматиращия низ, в случая **name**. Следва **%d,** което означава, да се замести с първото целочислено число подадено в аргументите. Последният специален символ е отново **%s**, което означава да се замести със следващия по ред символен низ (**town**). Следва **\n**, което е специален символ, който указва минаване на нов ред.

Форматиращи низове

Както вече казахме, методите на **System.out.printf** използват **форматните спецификатори** (**format specifiers**) съставящи **форматиращия низ**.

```
printf("…<format specifier>…<format specifier>…", <parameters>)
```

Форматиращите низове притежават мощен контрол върху показваната стойност и за това могат да придобият сложен вид. Следващата схема показва генералния синтаксис на **форматните спецификатори**.

```
%[argument_index$][flags][width][.precision]conversion
```

Както забелязваме първо се поставя специалния символ процент (**%**), който задава започването на форматиращ спецификатор. Ще обясним всеки от форматните спецификатори, след което ще разгледаме още примери. Всички аргументи са незадължителни освен **conversion.**

- **argument_index** - целочислен тип показващ позицията на аргумента от "аргументния списък". Първият аргумент се индикира с "**1$**", втория с "**2$**" и т.н.
- **flags** - поредица от символи които модифицират изходния формат. (Примерни ефекти са, показването на числото да бъде винаги със знак, слагането на скоби на отрицателните числа и т.н.)
- **width -** неотрицателен целочислен тип показващ минималния брой символи, които трябва да се отпечатат.
- **precision -** неотрицателен целочислен тип ограничаващ броя на показваните символи. Този атрибут си променя поведението според

`conversion` спецификатора. В случай на `float` определя броя цифри след запетаята.

- **conversion** – символ, показващ начина на конвертиране на аргумента. Набора от валидни конверсии зависи от типа.

Ще разгледаме таблици с валидните стойности за всеки един от форматните спецификатори. В първата колона ще показваме стойност, а втората колона ще обяснява какъв ще е изхода при прилагане на съответната стойност.

Стойности за форматния спецификатор

Ще покажем таблица с най-често използваните спецификатори и техни възможни стойности.

Конвертиране (conversion)

В таблицата са дадени някои форматни спецификатори и тяхното значение:

Конвертиране		Описание
`b`		Булева стойност (`true, false`)
`o`		Осмична стойност
`x`		Шестнадесетична стойност
`c`		Знаков тип (`char`)
`s`		Стринг
`S`		Стринг, форматиран с главни букви
`f`		Число с плаваща запетая (`float, double`)
`e`		Число с плаваща запетая (с голяма точност)
`h`		Хеш кода на аргумента в шестнадесетичен вид
`n`		Нов ред "`%n`", е еквивалентно на "`\n`"
`t`		Префикс за конвертиране към дата. Тази конверсия се използва само в комбинация с някоя от долните опции. Пример: $tH, $tM, $tS
`t`	`Y`	Година, форматирана с 4 цифри (2010)
`t`	`y`	Година, форматирана с 2 цифри (10)
`t`	`m`	Месец в интервал: 01 – 13
`t`	`B`	Месец, като стринг зависещ от локализацията (January)
`t`	`b`	Месец, като "кратък" стринг, зависещ от локализацията
`t`	`A`	Ден от седмицата, като стринг ("Sunday", "Monday")
`t`	`a`	Ден от седмицата, като "кратък" стринг ("Sun", "Mon")
`t`	`d`	Ден от месеца в интервал: 01 – 31

t	j	Ден от годината в интервал: 000 – 366
t	H	Час в интервал: 00 – 23
t	l	Час в интервал: 0 – 12
t	M	Минути в интервал: 00 – 59
t	S	Секунди във формат: 00 – 60
t	L	Милисекунди във формат: 000 – 999
t	N	Наносекунди във формат: 000000000 – 999999999
t	p	Спецификатор на деня. Пример: "am" или "pm"

Флагове (flags)

В таблицата са дадени някои флагове и тяхното действие:

Конверсия	Описание
-	Резултатът ще бъде ляво ориентиран
+	Резултатът винаги ще включва знак (+, -)
0	Резултатът ще се отмести с нули
(	Резултатът ще затвори в скоби отрицателните числа

Форматиращи низове – примери

Ще разгледаме основни примери, използващи форматиране на стрингове. За всеки от примерите изхода е описан чрез "**// Output:** ", както и кратки разяснителни коментари поставени в скоби.

PrintingFormattedStrings.java

```
public class PrintingFormattedStrings {

   public static void main(String[] args) {

      String name = "Boris";
      int age = 18;
      String town = "Plovdiv";

      System.out.printf("My name is %s. \n", name);
      // Output:   My name is Boris.

      System.out.printf("My name is %S. \n", name);
      // Output:   My name is BORIS. (name in uppercase "%S")

      System.out.printf(
         "%1$s is big town.\n" +
         "%2$s lives in %1$s.\n" +
         "%2$s is %3$d years old.\n",
```

```
        town, name, age);

    // Output:   Plovdiv is big town.
    //           Boris lives in Plovdiv.
    //           Boris is 18 years old.

    int a = 2, b = 3;
    System.out.printf("%d + %d =", a, b);
    System.out.printf(" %d\n", (a + b));
    // Output: 2 + 3 = 5 (two prints without new line)

    System.out.printf("%d * %d = %d%n", a, b, a * b);
    // Output: 2 * 3 = 6 (with new line at end "%n")

    float pi = 3.14159206f;
    System.out.printf("%.2f%n", pi);
    // Output: 3,14 (using [.precision] = 2)

    System.out.printf("%.5f%n", pi);
    // Output: 3,14159  (using [.precision] = 5)

    double colaPrice = 1.20;
    String cola = "Coca Cola";
    double fantaPrice = 1.20;
    String fanta = "Fanta Bamboocha";
    double kamenitzaPrice = 1.50;
    String kamenitza = "Kamenitza";

    System.out.println("\nMenu:");
    System.out.printf("1. %s – %.2f%n", cola, colaPrice);
    System.out.printf("2. %s – %.2f%n", fanta, fantaPrice);
    System.out.printf("3. %s – %.2f%n",
      kamenitza, kamenitzaPrice);
    System.out.println();
    // Output: Menu:
    //           1. Coca Cola – 1,20
    //           2. Fanta Bamboocha – 1,20
    //           3. Kamenitza – 1,50

    System.out.println("Next sentence will be" +
       " on a new line.");

    System.out.printf("Bye, bye, %s from %s.%n", name, town);

    // Output: Next sentence will be on a new line.
    //           Bye, bye, Boris from Plovdiv.
  }
}
```

Форматиране на дати и часове – примери

Следващият пример показва форматиране на дати и часове:

PrintingFormattedDates.java

```
public class PrintingFormattedDates {

   public static void main(String[] args) {

      System.out.printf("The time is: %1$tH:%1$tM:%1$tS.\n",
           new java.util.Date());
      // The time is: 13:54:36. (ends with new line "\n")

      Date date = new Date();
      System.out.printf("The date in Day/Month/Year is:
           %1$td/%1$tm/%1$tY.\n", date);
      // The date in Day/Month/Year is: 09/08/2008.

      System.out.printf("The date and time is: %1$tA
           %1$tI:%1$tM%1$tp %1$tB/%1$tY. \n", date);
      // The date and time is: Събота 05:08pm Август/2008.
   }
}
```

Форматиране на числа и дати – още примери

Ето и още примери за печатане на конзолата чрез форматиращи низове:

MoreExamplesWithFormatting.java

```
public class MoreExamplesWithFormatting {

   public static void main(String[] args) {

      long n = 120582;
      System.out.format("%d%n", n);          //  -->  "120582"
      System.out.format("%08d%n", n);        //  -->  "00120582"
      System.out.format("%+8d%n", n);        //  -->  " +120582"
      System.out.format("%,8d%n", n);        //  -->  " 120,582"
      System.out.format("%+,8d%n%n", n);     //  -->  "+120,582"

      double pi = Math.PI;
      System.out.format("%f%n", pi);         //  -->  "3.141593"
      System.out.format("%.3f%n", pi);       //  -->  "3.142"
      System.out.format("%10.3f%n", pi);     //  -->  "     3.142"
      System.out.format("%-10.3f%n", pi); //  -->  "3.142"
      System.out.format(Locale.ITALIAN,
         "%-10.4f%n%n", pi);                 //  -->  "3,1416"
```

```
        Calendar c = Calendar.getInstance();
        System.out.format("%tB %te, %tY%n", c, c, c);
                                            //  -->  "Август 9, 2008"

        System.out.format("%tl:%tM %tp%n", c, c, c);
                                            //  -->  "5:29 pm"
    }
}
```

Форматиращи низове и локализация

При използването на форматиращи низове е възможно една и съща програма да отпечатва различни стойности в зависимост от настройките за локализация в операционната система. Например, при отпечатване на месеца от дадена дата, ако текущата локализация е българската, ще се отпечата на български, примерно "Август", докато ако локализацията е американската, ще се отпечата на английски, примерно "August".

При стартирането си Java виртуалната машина автоматично извлича системната локализация на операционната система и ползва нея за четене и писане на форматирани данни (числа, дати и други).

Локализацията може да се променя ръчно чрез класа **java.util.Locale**. Ето един пример, в който отпечатваме едно число и текущата дата и час по американската и по българската локализация:

```
Locale.setDefault(Locale.US);
System.out.println("Locale: " + Locale.getDefault().toString());
System.out.printf("%.2f\n", 1234.56);
System.out.printf("%1$tA %1$tI:%1$tM%1$tp %1$tB-%1$tY.\n\n",
   new Date());

Locale.setDefault(new Locale("bg", "BG"));
System.out.println("Locale: " + Locale.getDefault().toString());
System.out.printf("%.2f\n", 1234.56);
System.out.printf("%1$tA %1$tH:%1$tM %1$tB-%1$tY.\n",
   new Date());
```

Резултатът от изпълнението на този код е следният:

```
Locale: en_US
1234.56
Saturday 05:24pm November-2008.

Locale: bg_BG
1234,56
Събота 17:24 Ноември-2008.
```

Вход от конзолата

Както в началото на темата обяснихме, най-подходяща за малки приложения е конзолната комуникация, понеже е най-лесна за имплементиране. **Стандартното входно устройство** е тази част от операционната система, която контролира от къде програмата ще получи своя вход. По подразбиране "стандартното входно устройство" чете своя вход от драйвер "закачен" за клавиатурата. Това може да бъде променено и стандартният вход може да бъде пренасочен към друго място, например към файл.

Всеки език за програмиране има механизъм за четене и писане в конзолата. Обектът, контролиращ стандартния входен поток в **Java,** е **System.in**.

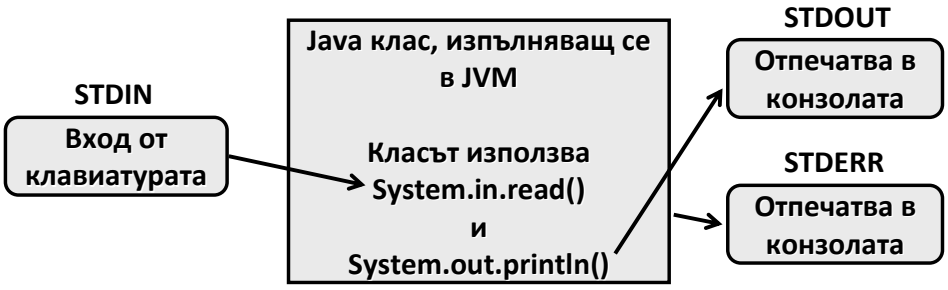
От конзолата можем да четем различни данни:

- текст;
- числени типове, след конвертиране.

Четене от потока System.in

Да си припомним, че системният клас **System** има статична член-променлива **in**. Тази член променлива е от тип **InputStream**. Това е базов клас (предшественик) на всички класове, представящи входен поток от байтове. Методите на този клас и най-вече **System.in.read()** се използват за четене от стандартния вход.

В чистия му вариант **System.in.read()** почти не се използва, защото има абстракции, които улесняват четенето от конзолата. В последните версии на Java се появяват допълнителни улеснения за четене, но идеологията винаги си остава една и съща.



Тази схема показва взаимодействието между трите стандартни променливи предназначени за вход (**STDIN**) и изход (**STDOUT, STDERR**) в **Java**. **STDERR** е стандартният изход за грешки. На него може да печатаме по следния начин:

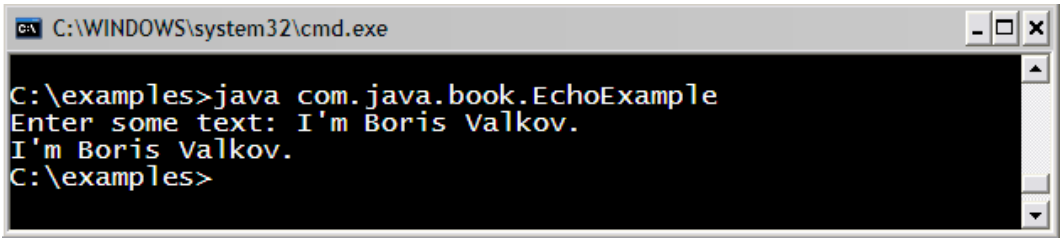
```
System.err.println("This is error!");
```

Ще разгледаме примери, четящи през **System.in.read()**, за да може да разберем по-детайлно процеса по четене, след което ще разгледаме и улесненията предоставени от новите версии на Java. Ето един пример, в който четем директно от **System.in**:

EchoExample.java

```
public class EchoExample {

   public static void main (String [] args)
      throws java.io.IOException {

      int ch;
      System.out.print ("Enter some text: ");
      while ((ch = System.in.read()) != '\n') {
         System.out.print((char) ch);
      }
   }
}
```

Резултатът от изпълнението на примера е следният:



Най-общо тази програма чете текст, въведен от потребителя, и го принтира в конзолата. Ще разгледаме програмата по-подробно.

Първо извеждаме помощен текст в конзолата ("Enter some text:"), за да уведомим потребителя какво точно трябва да въведе. След това извикваме в цикъл **System.in.read()**. Четенето чрез метода **read()** става байт по байт. Влизаме в безкрайно четене (**while loop**), докато не се натисне клавиша за край на ред ("\n"). Ако този клавиш се натисне, четенето прекратява и се изписва прочетеното. Получава се нещо като "ехо".

Четенето на символ започва при въведен край на ред

Изходът от програмата изглежда малко странно. Нали след като прочетем даден символ веднага го отпечатваме и след това веднага четем следващ символ и т.н. Би трябвало след всеки прочетен символ да се отпечатва същия символ, но това не се случва. Въведените символи се прочитат наведнъж, след натискане на [Enter] и след това се отпечатват наведнъж. Причината за това е, че четенето от конзолата в Java става ред по ред, а не символ по символ. Макар и да имаме метод за четене на един символ, той изчаква въвеждането на цял ред и натискането на [Enter] и чак тогава започва четенето на въведените до момента символи, които се натрупват в специален буфер. Ако не знаем тази особеност, може дълго да се чудим защо горната програма има такова странно поведение.

Методът System.in.read()

Метода **read()** не приема аргументи и връща **int**. Методът връща 7-битов ASCII код (ако стандартното входно устройство е клавиатура) или 8-битов байт (ако стандартното входно устройство е пренасочено към файл). И в двата случая **System.in.read()** превръща числото в **32-bit integer** и връща резултата.

Например при операционна система Windows, когато стандартното входно устройство е клавиатура ще се случи следното: Когато натискаме клавиши от клавиатура "контролирана" от Windows, операционната система съхранява натиснатите клавиши като **7-bit ASCII** код във вътрешен "клавишен буфер". Този буфер може да съхранява приблизително **16 ASCII** кода и е организиран, като структура от данни "опашка" (първи влязъл – първи излязъл). **System.in.read()** взима **7-bit ASCII** код от главата на клавишния буфер и изтрива този код от буфера. Този **7-bit ASCII** код се конвертира до **int** от метода **System.in.read()**, като върнатите 7 бита се запълва с 25 празни бита, за да се получи 32-битовия **int** който се връща от метода **System.in.read()**. Ако имаме "cast" към **char**, тогава този 32-битов **int** се конвертира до 16-битов **Unicode**, каквото е представянето на **char** в Java. Повторно извикване на **System.in.read()** взима следващия **ASCII** код и така нататък.

Ако се окаже, че в клавишния буфер няма никакви **ASCII** кодове, които **System.in.read()** да прочете, методът блокира и минава в режим на изчакване докато не дойде следващ **ASCII** код. Програмите използващи метода **System.in.read()** за четене от конзола трябва да проверяват дали не е натиснат клавишът за край на ред. В Windows това е клавишът [Enter]. При натискането му, Windows слага в буфера "carriage return" код (**ASCII 13**) следван от "new-line" код (**ASCII 10**).

Класът Scanner

Този клас е въведен от Java 5.0 насам, за да улесни четенето на текст и числови данни от конзолата, както и от файлове. Класът представлява опростяване (абстракция) на сложното до скоро четене през потоци и буфери. Класът **java.util.Scanner** има различни методи улесняващи форматирания вход от конзолата:

- **nextInt()** / **nextLong()** за четене на целочислени типове
- **nextFloat() / nextDouble()** за четене на числа с плаваща запетая
- **nextLine()** за четене на цял символен низ, връща **String**
- **hasNext***()** проверява дали съществува съответния тип (**long**, **float**, …)

Всеки от методите може да хвърли **IllegalStateException**, ако **Scanner** класът е вече затворен "**scanner.close()**". Подробно на изключенията, като

начин за известяване за грешка или друг проблем ще се спрем в главата "Обработка на изключения".

Всеки от методите **next***()** може да хвърли **NoSuchElementException** ако входния поток свърши, но търсения елемент не съществува (**int, long, line**). В случая с четенето от конзолата входния поток не може да свърши, но в случай че четем от файл, при достигане на края на файла свършва и потокът, свързан с него. Повече за потоци и файлове ще научите в главата "Текстови файлове".

Методите, които очакват число, могат да предизвикат изключение **InputMismatchException**, ако това, което са получили, не може да се преобразува до желания числен тип.

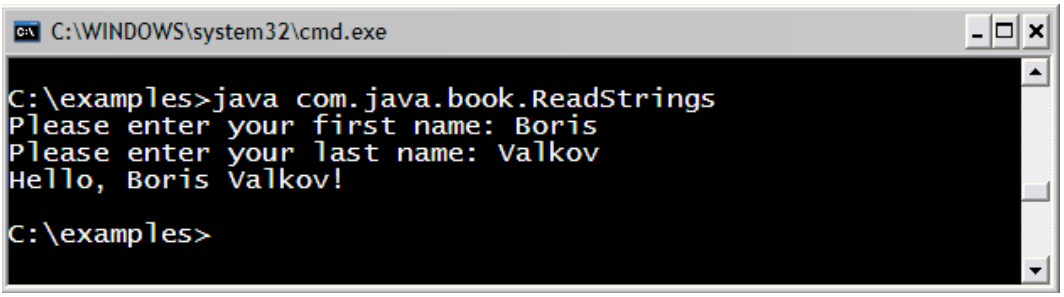
Четене на цял ред чрез BufferedReader

Първо ще разгледаме пример показващ стария подход за четене на ред от конзолата (преди въвеждането на класа **Scanner**):

ReadingStringsOldStyle.java

```
public class ReadStringsOldStyle {

   public static void main(String[] args) throws IOException {

      // Open the standard input
      BufferedReader br = new BufferedReader(
         new InputStreamReader(System.in));

      System.out.print("Please enter your first name: ");
      String firstName = br.readLine();

      System.out.print("Please enter your last name: ");
      String lastName = br.readLine();

      System.out.printf("Hello, %s %s!\n", firstName, lastName);

      // br.close(); - Do not close stream reading System.in!
   }
}
```

Резултатът от изпълнението на примера може да изглежда по следния начин:



Няма да се спираме подробно на този пример, понеже има по-лесни начини за четене от конзолата. Можем само да забележим, че има "навързване" на потоци и буферен четец. Потокът **InputStreamReader** представлява "мост" между байтови (какъвто е **InputStream** от **System.in**) и символни потоци. Обикновено потокът **InputStreamReader** се обвива от **BufferedReader** с цел улесняване на четенето. Чрез **BufferedReader** може да се чете цял ред от символи наведнъж, без да е нужно да се пишат цикли (както в предишния пример).

Четене на цял ред чрез Scanner

Следващият пример е аналогичен, но ще използваме класа **Scanner**:

ReadingStringsNewStyle.java

```
public class ReadingStringsNewStyle {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Please enter your first name: ");
      String firstName = input.nextLine();

      System.out.print("Please enter your last name: ");
      String lastName = input.nextLine();

      System.out.printf("Hello, %s %s!\n", firstName, lastName);

      // input.close(); - Don't close Scanner reading System.in!
   }
}

// Output: Please enter your first name: Boris
//           Please enter your last name: Valkov
//           Hello, Boris Valkov!
```

Виждаме колко лесно става четенето на текст от конзолата с класа **`Scanner`**. Голяма част от сложността и недостатъците на старите подходи са скрити зад този клас.

- Първо създаваме един обект от тип **`java.util.Scanner`**. В конструктора на този обект подаваме "стандартния входен поток" (**`System.in`**) от който ще четем.
- Отпечатваме текст в конзолата, който пита за името на потребителя.
- Извършваме четене на цял ред от конзолата, чрез метода **`nextLine()`**.
- Повтаряме горните две стъпки за фамилията.
- След като сме събрали необходимата информация я отпечатваме в конзолата.
- В случай когато ресурсът е файл или нещо друго (**`Readable`**), след приключване на работа с класа **`Scanner`** го затваряме с метода **`close()`**, който от своя страна има отговорността да извика **`close()`** на подадения му в конструктора поток (или **`Closeable`**). **Обаче** в примера, който разгледахме, използваме **системен ресурс** **`System.in`**! Особеното за системните ресурси, е, че те "надживяват" кода на нашата програма и отговорността за тяхното отваряне и затваряне се поема от системен код, опериращ преди и след нашата програма. Това е причината **`close()`** методът в примерите да е коментиран, понеже не трябва да се извиква.

При четенето от конзолата има една особеност: тя не трябва да бъде изрично нито отваряна, нито затваряна, защото това се прави от операционната система.

В конструктора на `Scanner` се подава `System.in`. Както вече обяснихме, когато се извика `close()` на класа `Scanner`, по спецификация се затваря потокът, от който се чете, но това е стандартният вход (`System.in`) и ако той бъде затворен, вече няма да може да се чете от него. Следващи опити за четене от `System.in` ще доведат до грешки! Те могат да бъдат различни изключения, свързани с четене от затворен поток: `IOException`, `IllegalStateException` или `NoSuchElementException`.

Обработката на изключения е тема на една от следващите глави и засега няма да ви обърквам допълнително с тази трудна за начинаещите материя. В примерите по-горе за простота игнорираме евентуално възникващите грешки.

В почти никоя книга за програмиране няма да видите коректна обработка на изключения. Освобождаването на използваните ресурси също не винаги е дадено правилно.

	Една от причините за това е запазване на кода лесен за четене. Затова, НЕ копирайте код от учебни примери директно във вашата продукционна програма! **В реални програми, правилно обработване на изключения и пълно освобождаване на ресурси често са подценявани! Това е много тежка част от програмирането, в която често ГРЕШАТ и най-добрите!**

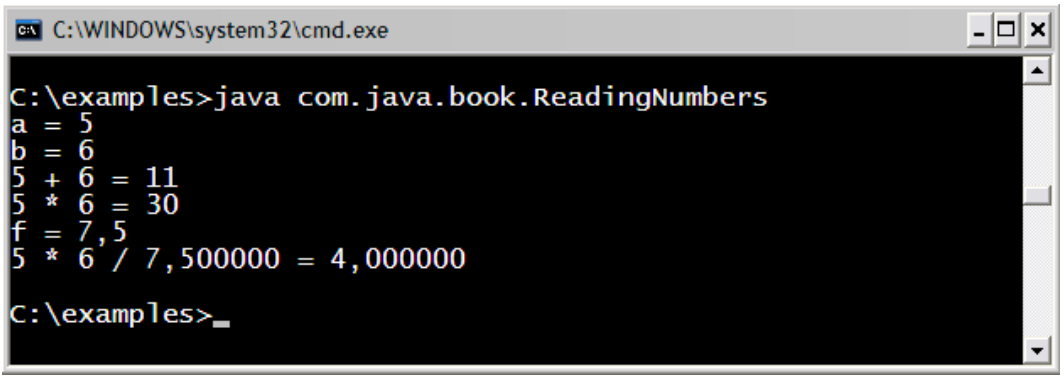
Четене на числа

Четенето на числа с класа **Scanner** е също толкова лесно, колкото и четенето на цял ред. Ето един пример:

ReadingNumbers.java

```
public class ReadingNumbers {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("a = ");
      int a = input.nextInt();

      System.out.print("b = ");
      int b = input.nextInt();
      // Output: a = 5
      //           b = 6

      System.out.printf("%d + %d = %d%n", a, b, a + b);
      System.out.printf("%d * %d = %d%n", a, b, a * b);
      // Output: 5 + 6 = 11
      //           5 * 6 = 30

      System.out.print("f = ");
      float f = input.nextFloat();
      System.out.printf("%d * %d / %f = %f%n",
         a, b, f, a * b / f);
   }
}
```

Резултата от изпълнението на програмата би могъл да е следният (при условие че въведем 5; 6 и 7.5 като входни данни):



Четене на число през класа **Scanner** става аналогично на четене на стринг. В този пример се четат две числа от конзолата, след което се извършват различни математически операции с тях и резултатите се отпечатват на екрана с форматирания изход (който разгледахме в началото на темата).

В този пример особеното е, че използваме методите за четене на числени типове и при грешно подаден резултат (например текст), ще възникне грешка (изключение) **InputMismatchException**. Това важи с особена сила при четенето на реално число, защото разделителят, който се използва между цялата и дробната част, не е строго фиксиран.

Разделителят за числата с плаваща запетая зависи от текущите езикови настройки на операционната система (Regional and Language Options в Windows). При едни системи за разделител може да се счита символът запетая, при други точка. Въвеждането на точка вместо запетая ще предизвика `InputMismatchException`.

Вход и изход на конзолата – примери

Ще разгледаме още няколко примера за вход и изход от конзолата.

Печатане на писмо

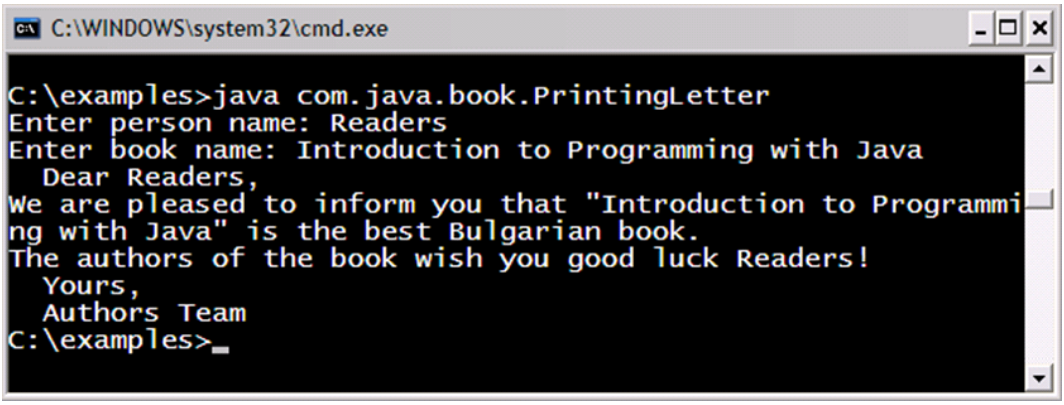
Това е един практичен пример показващ конзолен вход и форматиран текст под формата на писмо.

PrintingLetter.java

```
public class PrintingLetter {

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      System.out.printf("Enter person name: ");
      String person = input.nextLine();
```

```
        System.out.printf("Enter book name: ");
        String book = input.nextLine();

        String from = "Authors Team";

        System.out.printf("  Dear %s,%n", person);
        System.out.printf("We are pleased to inform " +
           "you that \"%2$s\" is the best Bulgarian book. \n" +
           "The authors of the book wish you good luck %s!%n",
           person, book);

        System.out.println("  Yours,");
        System.out.printf("  %s", from);
    }
}
```

Резултатът от изпълнението на горната програма би могъл да следния:



В този пример имаме предварителен шаблон на писмо. Програмата "задава" няколко въпроса на потребителя и по този начин прочита от конзолата нужната информация, за да отпечата писмото, като замества форматиращите спецификатори с попълнените от потребителя параметри. Следва печатане на цялото писмо в конзолата.

Лице на правоъгълник или триъгълник

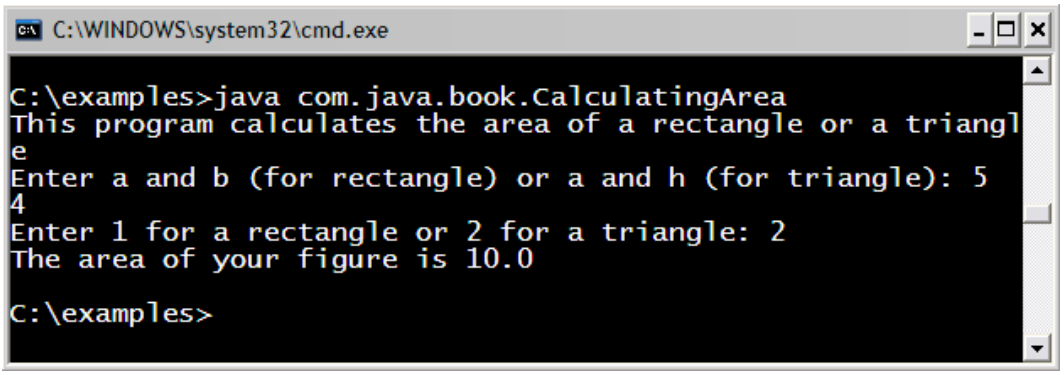
Ще разгледаме още един пример: изчисляване на лице на правоъгълник или триъгълник.

CalculatingArea.java

```
public class CalculatingArea {

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
```

```
        System.out.println("This program calculates " +
           "the area of a rectangle or a triangle");

        System.out.print("Enter a and b (for rectangle) " +
           "or a and h (for triangle): ");

        int a = input.nextInt();
        int b = input.nextInt();

        System.out.print("Enter 1 for a rectangle or " +
             "2 for a triangle: ");

        int choice = input.nextInt();
        double area = (double) (a * b) / choice;
        System.out.println("The area of your figure is " + area);
    }
}
```

Резултатът от изпълнението на горния пример е следният:



Упражнения

1. Напишете програма, която чете от конзолата три числа от тип **int** и отпечатва тяхната сума.
2. Напишете програма, която чете от конзолата радиуса "**r**" на кръг и отпечатва неговия периметър и обиколка.
3. Дадена фирма има име, адрес, телефонен номер, факс номер, уеб сайт и мениджър. Мениджърът има име, фамилия и телефонен номер. Напишете програма, която чете информацията за компанията и нейния мениджър и я отпечатва след това на конзолата.
4. Напишете програма, която чете от конзолата две цели числа (**integer**) и отпечатва, колко числа има между тях, такива, че остатъкът им от деленето на 5 да е 0.

5. Напишете програма, която чете от конзолата две цели числа и отпечатва по-голямото от тях. Реализирайте програмата без използването на сравнение. Забележка: задачата изисква малко да помислите!
6. Напишете програма, която чете пет числа и отпечатва тяхната сума.
7. Напишете програма, която чете пет числа и отпечатва най-голямото от тях. Забележка: трябва да използвате конструкция "**if**", която до момента не сме разгледали.

Решения и упътвания

1. Използвайте класа **Scanner**.
2. Използвайте константата **Math.PI** и добре известните формули от планиметрията.
3. Форматирайте текста с **printf(…)** подобно на този от примера с писмото, който разгледахме.
4. Има два подхода за решаване на задачата:

   Първи подход: Използват се математически хитрини за оптимизирано изчисляване, базирани на факта, че всяко пето число се дели на 5.

   Вторият подход е по-лесен, но работи по-бавно. Чрез **for** цикъл може да се обиколи и провери всяко число в дадения интервал. За целта трябва да прочетете от Интернет или от главата "Цикли" как се използва for цикъл.
5. Нека числата са a и b. Използвайте следните преобразувания: a=a-b; b=b+a; a=b-a.
6. Можете да прочетете чистата в пет различни променливи и накрая да ги сумирате. За да няма повторение на код, можете да разгледате конструкцията за цикъл "**for**" от главата "Цикли".
7. Трябва да използвате конструкцията за сравнение "**if**", за която можете да прочетете в Интернет или от следващите глави на книгата. За да избегнете повторението на код, можете да използвате конструкцията за цикъл "**for**", за която също трябва да прочетете в книгата или в Интернет.

Глава 5. Условни конструкции

Автор

Марин Георгиев

В тази тема...

В настоящата тема ще разгледаме условните конструкции в Java, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Ще обясним синтаксиса на условните оператори: **`if`** и **`if-else`** с подходящи примери и ще разясним практическото приложение на оператора за избор **`switch`**.

Ще обърнем внимание на добрите практики, които е нужно да бъдат следвани, с цел постигане на по-добър стил на програмиране при използването на вложени и други видове условни конструкции.

Оператори за сравнение и булеви изрази

В следващата секция ще опишем операторите за сравнение в езика Java. Те са важни, тъй като чрез тях можем да описваме условия при използване на условни конструкции.

Оператори за сравнение

В тази секция ще бъдат разгледани шест оператора за сравнение: `<`, `<=`, `>`, `>=`, `==` и `!=`. Операторите за сравнение винаги дават като резултат булева величина (**`true`** или **`false`**).

Java има няколко оператора за сравнение, които могат да бъдат използвани за сравняване на всяка комбинация от цели числа, числа с плаваща запетая или символи:

Оператор	Действие
`==`	равно
`!=`	различно
`>`	по-голямо
`>=`	по-голямо или равно

<	по-малко
<=	по-малко или равно

Нека погледнем към следните сравнения:

```
int weight = 700;
System.out.println(weight >= 500);

char sex = 'm';
System.out.println(sex <= 'f');

double colorWaveLength = 1.630;
System.out.println(colorWaveLength > 1.621);
```

В дадения програмен код използваме сравнение между числа и между символи. При сравнението на символи се сравнява тяхната лексикографска наредба (сравняват се Unicode номерата на съответните символи). Стартирайки примера ще получим следния резултат:

```
true
false
true
```

В Java има четири различни типа данни, които могат да бъдат сравнявани:

- Числа (**int**, **long**, **float**, **double**, …)
- Символи (**char**)
- Булеви стойности (**boolean**)
- Референции към обекти, познати още като обектни указатели (**String**, **Object** и други)

Всяко едно сравнение може да засегне две числа (включително **char**), две **boolean** стойности, или две референции към обекти. По принцип не е забранено да се сравнява символ с число, но това не се препоръчва, защото води до труден за четене и разбиране код.

Сравнение на цели числа и символи

Програмният код по-долу представя някои тестове за сравняване на числа и символи:

```
System.out.println("char 'a' == 'a'? " + ('a' == 'a'));
System.out.println("char 'a' == 'b'? " + ('a' == 'b'));
System.out.println("5 != 6? " + (5 != 6));
System.out.println("5.0 == 5L? " + (5.0 == 5L));
System.out.println("true == false? " + (true == false));
```

Този програмен код извежда следния резултат:

```
char 'a' == 'a'? true
char 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false
```

Сравнение на реални типове

Както можем да забележим, ако число с плаваща запетая се сравнява с цяло число, и стойностите и на двете са едни и същи, операторът == връща **true** (истина), както се и очаква. Това, обаче не винаги работи коректно. Ето един любопитен пример:

```
float value = 1.0f;
System.out.println(value);
float sum = 0.1f + 0.1f + 0.1f + 0.1f + 0.1f +
   0.1f + 0.1f + 0.1f + 0.1f + 0.1f;
System.out.println(sum);
System.out.println(sum == value);
```

Резултатът от изпълнение на горния код е абсолютно неочакван:

```
1.0
1.0000001
false
```

Оказва се, сумата от 10 пъти 1/10 не е равно на 1. При работата с реални числа не всяко число има точно представяне в типовете **float** и **double** и поради закръглянето се получават грешки. По тази причина често пъти сравнението на реални числа се прави с някаква точност, например 0.000001. Счита се, че две реални числа са равни, ако имат много малка разлика по абсолютна стойност. Ето пример за сравнение на реални числи с точност:

```
float value = 1.0f;
float sum = 0.1f + 0.1f + 0.1f + 0.1f + 0.1f +
   0.1f + 0.1f + 0.1f + 0.1f + 0.1f;
System.out.println("Exact compare: " +
   (sum==value));
System.out.println("Rounded compare: " +
   (Math.abs(sum-value) < 0.000001));
```

Резултатът от горния код показва, че сравнението с точност работи по-добре при реални числа, отколкото точното сравнение:

```
Exact compare: false
```

```
Rounded compare: true
```

Сравнение на референции към обекти

Два указателя към обекти (референции) могат да сочат към един и същи обект. Това е показано в следния програмен код:

```
String str = "Some text";
String anotherStr = str;
```

След изпълнението на този код, двете променливи **str** и **anotherStr** ще сочат към един и същи обект (обект **String** със стойност "Some text").

Променливите от тип референция към обект могат да бъдат проверени, дали сочат към един и същ обект, посредством оператора за сравнение **==**. Този оператор не сравнява съдържанието на обектите, а само дали се намират на едно и също място в паметта, т. е. дали са един и същ обект. За променливи от тип обект, не са приложими сравненията по големина (**<**, **>**, **<=** и **>=**).

За да разберем разликата между сравнение на референции към обекти (адреси на обекти в паметта) и сравнение на стойности на обекти, нека следния програмен код:

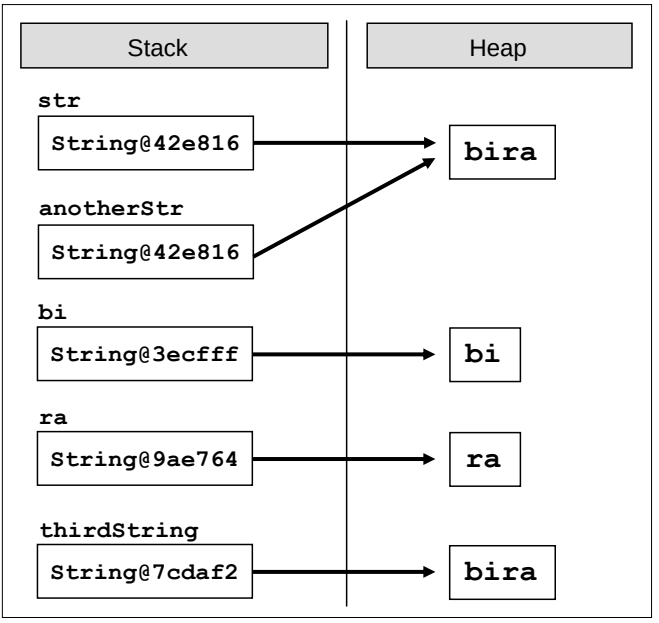
```
String str = "bira";
String anotherStr = str;
String bi = "bi";
String ra = "ra";
String thirdStr = bi  + ra;

System.out.println("str = " + str);
System.out.println("anotherStr = " + anotherStr);
System.out.println("thirdStr = " + thirdStr);
System.out.println(str == anotherStr);
System.out.println(str == thirdStr);
```

Ако изпълним примера, ще получим следния резултат:

```
str = bira
anotherStr = bira
thirdStr = bira
true
false
```

В примера се създават три променливи, които съдържат една и съща стойност "**bira**". Първите две от тях са референции към един и същ обект в паметта, т.е. са еднакви указатели (адреси в паметта). Третият обект, обаче, се намира на друго място в паметта, макар че има същата стойност

като другите два. За да си представите визуално това ще ви помогне фигурата:



Повече за класа **String** и за сравняването на символните низове ще научите в главата "Символни низове".

Логически оператори

В настоящата секция ще разгледаме логическите оператори за сравнение. Те биват шест вида: **&**, **|**, **^**, **!**, **&&** и **||**.

Логически оператори && и ||

Тези логически оператори се използват само върху **boolean** стойност. За да бъде резултатът от сравняването на два израза с оператор **&& true** (истина), то и двата операнда трябва да имат стойност **true**. Например:

```
if ((2 < 3) && (3 < 4)) {

}
```

Този израз е истина, когато и двата операнда: (2 < 3) и (3 < 4) са **true**. Логическият оператор **&&** се нарича още и съкратен оператор, тъй като той не губи време за допълнителни изчисления. Той изчислява лявата част на израза (първи операнд) и ако резултатът е **false**, то операторът **&&** не губи време за изчисляването на втория операнд, тъй като е невъзможно крайният резултат да е "истина", ако и двата операнда не са "истина". По тази причина той се нарича още **съкратен логически оператор "и"**. Ето един пример:

```
class Logical {
   public static void main(String[] args) {
      boolean b = true && false;
      System.out.println("b = " + b);
   }
}
```

Когато изпълним предходния програмен код, получаваме като резултат:

```
b = false
```

Операторът || е подобен на && с тази разлика, че той следи поне един от изчисляваните операнди да е "истина". Той се нарича още **съкратен логически оператор "или"**, тъй като ако изчисляването на първия операнд се свежда до "истина", не губи време за изчисление на втория операнд, защото резултатът във всички случаи ще бъде "истина".

Ако при изчислението се получи така, че и двата операнда се свеждат до "лъжа", то крайният резултат от сравнението ще бъде "лъжа".

Логически оператори & и |

Операторите за сравнение & и | са подобни, съответно на && и ||. Разликата се състои във факта, че се изчисляват и двата операнда един след друг, независимо от това, че крайния резултат е предварително ясен. Затова и тези оператори за сравнение се наричат още **несъкратени логически оператори**.

Например, когато се сравняват два операнда със & и първият операнд се сведе до "лъжа" въпреки това се продължава с изчисляването на вторият операнд. Резултатът е ясно, че ще бъде сведен до "лъжа". По същия начин, когато се сравняват два операнда със | и първия операнд се сведе до "истина", независимо от това се продължава с изчисляването на втория операнд и резултатът въпреки всичко се свежда до "истина".

Логически оператори ^ и !

Операторът ^, известен още като изключващо ИЛИ (XOR), се прилага само върху булеви стойности. Той се причислява към несъкратените оператори, поради факта, че изчислява и двата операнда един след друг. Резултатът от прилагането на оператора е истина, когато само и точно един от операндите е истина, но не и двата едновременно. В противен случай резултатът е лъжа. Ето един пример:

```
System.out.println("Изключващо ИЛИ: " + ((2<3) ^ (4>3)));
```

Резултатът е следният:

```
Изключващо ИЛИ: false
```

Предходният израз е сведен до лъжа, защото и двата операнда: (2 < 3) и (4 > 3) са истина.

Операторът **!** връща като резултат противоположната стойност на булевия израз, към който е приложен. Пример:

```
boolean value = ! (7 == 5); // true
```

Горният израз може да бъде прочетен, като "обратното на истинността на израза 7 == 5". Резултатът от примера е **true** (обратното на **false**).

Побитови операции върху цели числа

Три от логическите оператори, споменати по-горе могат да оперират не само върху булеви изрази, но и върху числови стойности. В този случай те се използват за извършване на побитови операции върху цели числа. Това са операторите: **&**, **|** и **^**. Нека разгледаме следния програмен код:

```
byte b1 = 6 & 5; // 00000110 & 00000101 = 00000100
byte b2 = 7 | 9; // 00000111 | 00001001 = 00001111
byte b3 = 5 ^ 4; // 00000101 ^ 00000100 = 00000001
System.out.println(b1 + " " + b2 + " " + b3);
```

Резултатът от изпълнението му е:

```
4  15  1
```

Побитовите оператори сравняват две променливи бит по бит (в тяхното двоично представяне като поредици от нули и единици) и връщат като резултат променлива, чийто битове се получават по следния начин:

- При сравняване с побитовия оператор **&**: текущо върнатият бит е "вдигнат" или 1 когато и двата текущо сравнявани бита на двете променливи са 1).
- При сравняване с побитовия оператор **|**: текущо върнатият бит е "вдигнат" или 1 когато единия или и двата текущо сравнявани бита на двете променливи е 1).
- При сравняване с побитовия оператор **^**: текущо върнатият бит е "вдигнат" или 1 когато точно единия от двата текущо сравнявани бита на двете променливи е 1).

Условни конструкции if и if-else

Условните конструкции **if** и **if**-else представляват тип условен контрол, чрез който вашата програма може да се държи различно, в зависимост от приложения условен тест.

Условна конструкция if

Основният формат на условната конструкция **if** е, както следва:

```
if (булев израз) {
   тяло на условната конструкция
}
```

Форматът включва: **if**-клауза, булев израз и тяло на условната конструкция.

Булевият израз може да бъде променлива от булев тип, булев логически израз или релационен израз. Булевият израз не може да бъде цяло число.

Тялото е онази част, заключена между големите къдрави скоби: **{}**. То може да се състои от един или няколко оператора. Когато се състои от няколко оператора, говорим за съставен блоков оператор. Също така в тялото могат да бъдат включвани една или няколко конструкции.

Изразът в скобите трябва да бъде сведен до булева стойност **true** или **false**. Ако изразът бъде изчислен до стойност **true**, тогава се изпълнява тялото на условната конструкция. Ако пък от друга страна, резултатът от изчислението на булевия израз е **false**, то операторите в тялото няма да бъдат изпълнени.

Условна конструкция if – пример

Пример за условна конструкция **if**:

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter two numbers.");
   int firstNumber = input.nextInt();
   int secondNumber = input.nextInt();
   int biggerNumber = firstNumber;
   if (secondNumber > firstNumber) {
      biggerNumber = secondNumber;
   }
   System.out.printf("The bigger number is: %d%n", biggerNumber);
}
```

Конструкцията if и къдравите скоби

При наличието на само един оператор в тялото на **if**-конструкцията, къдравите скоби, обозначаващи тялото на условния оператор могат да бъдат изпуснати, както е показано по-долу. Добра практика е, обаче те да бъдат поставяни, дори при наличието на само един оператор. Целта е програмния код да бъде по-четим.

```
int a = 6;
```

```
if (a > 5)
   System.out.println("Променливата a е по-голяма от 5.");
   System.out.println("Този код винаги ще се изпълни!");
// Bad practice: unreadable code.
```

В горния пример кодът е форматиран заблуждаващо и създава впечатление, че и двете печатания по конзолата се отнасят за тялото на **if** блока, а всъщност това е вярно само за първия от тях.

Винаги слагайте къдрави скоби { } за тялото на if блоковете дори ако то се състои само от един оператор!

Условна конструкция if-else

Основният формат на условната конструкция **if-else** е, както следва:

```
if (булев израз) {
   тяло на условната конструкция
} else {
   тяло на else-конструкция
}
```

Форматът включва: запазена дума **if**, булев израз, тяло на условната конструкция, запазена дума **else**, тяло на **else**-конструкция. Тялото на **else**-конструкцията може да се състои от един или няколко оператора.

Изчислява се изразът в скобите (булевият израз). Резултатът от изчислението може да бъде сведен до **true** или **false**. В зависимост от резултата са възможни два пътя, по които да продължи потока от изчисления. Ако булевият израз се сведе до true, то се изпълнява тялото на условната конструкция, а тялото на **else**-конструкцията се пропуска и операторите в него не се изпълняват. От друга страна, ако булевият израз се сведе до **false**, то се изпълнява тялото на **else**-конструкцията, а тялото на условната конструкция се пропуска и операторите в него не се изпълняват.

Условна конструкция if-else – пример

Нека разгледаме следния програмен код:

```
x = 3;
if (x > 3) {
   System.out.println("x е  по-голямо от 3");
} else {
   System.out.println("x не е по-голямо от 3");
}
```

Програмният код може да бъде интерпретиран по следния начин: Ако x>3, то резултатът на изхода е: "x е по-голямо от 3", иначе (**else**) резултатът е: "x не е по-голямо от 3". В случая, понеже x=3, след изчислението на булевия израз ще бъде изпълнен операторът от **else**-конструкцията. Резултатът от примера е:

```
x не е по-голямо от 3
```

Вложени if конструкции

Понякога е нужно програмната логика в дадена програма или приложение да бъде представена посредством if-конструкции, които се съдържат една в друга. Наричаме ги **вложени if** или **if**-else конструкции.

Влагане наричаме поставянето на **if** или **if-else** клауза в друга **if** или **else** конструкция. Всяка **else** клауза се отнася за най-близко разположената предходна **if** клауза. По този начин разбираме коя **else** клауза към коя **if** клауза се отнася.

Не е добра практика нивото на влагане да бъде повече от три, тоест не трябва да бъдат влагани повече от три условни конструкции една в друга.

Ако поради една или друга причина се наложи да бъде направено влагане на повече от три конструкции, то трябва да се търси проблем в архитектурата на създаваната програма или приложение.

Вложени if конструкции – пример

Пример за употреба на вложени **if** конструкции:

```
Scanner input = new Scanner(System.in);
System.out.println(
  "Please enter two numbers (on separate lines).");
int first = input.nextInt();
int second = input.nextInt();
if (first == second) {
   System.out.println("These two numbers are equal.");
} else {
   if (first > second) {
      System.out.println("The first number is greater.");
   } else {
      System.out.println("The second number is greater.");
   }
}
```

В примера се въвеждат две числа и се сравняват на две стъпки: първо се сравняват дали са равни и ако не са, се сравняват отново, за да се установи кое от числата е по-голямо. Ето примерен резултат от работата на горния код:

```
Please enter two numbers (on separate lines).
2
4
The second number is greater.
```

Вложени if конструкции – добри практики

Подходи, които е препоръчително да бъдат следвани при писането на вложени **if** конструкции:

- Използвайте блокове, заградени с къдрави скоби **{ }** с цел избягване на двусмислие;
- Използвайте else клауза след всяка **if** клауза, когато е възможно;
- Поставяйте условните ситуации, които желаете да бъдат изпълнени първи на първо място;
- Форматирайте винаги програмния код, с цел да бъде лесно четим и да не позволява двусмислие;
- По-добре е използването на **switch-case** конструкция вместо вложени **if** конструкции, когато това е възможно;

Условна конструкция switch-case

В следващата секция ще бъде разгледана условната конструкция **switch** за избор измежду списък от възможности.

Как работи switch-case конструкцията?

Конструкцията switch прави избор измежду части от програмен код на базата на изчислената стойност на определен целочислен израз (целочислен селектор). Форматът на конструкцията за избор на вариант е:

```
switch (целочислен селектор) {
   case целочислена-стойност-1: конструкция; break;
   case целочислена-стойност-2: конструкция; break;
   case целочислена-стойност-3: конструкция; break;
   case целочислена-стойност-4: конструкция; break;
   // …
   default: конструкция;
}
```

Целочисленият селектор е израз, даващ като резултат целочислена стойност. Операторът **switch** сравнява резултата от целочисления селектор с всяка една целочислена стойност (етикет). Ако се открие съвпадение, се изпълнява съответната конструкция (проста или съставна). Ако не се открие съвпадение, се изпълнява **default** конструкцията. Стойността на целочисления израз трябва задължително да бъде изчислена преди да се сравнява с целочислените стойности вътре в **switch** конструкцията.

Виждаме, че в горната дефиниция всеки case завършва с **break**, което води до преход към края на тялото на **switch**. Това е стандартният начин за изграждане на **switch** конструкция, но **break** е незадължителна клауза. Ако липсва, кодът след **case** конструкцията, при която е срещнато съвпадение между стойността на селектора и на етикета, ще се изпълни и след това ще бъдат изпълнени конструкциите на другите case-оператори надолу, независимо че стойностите на техните етикети не биха съвпаднали със стойността на селектора. Това продължава докато бъде достигната break клауза след някоя case конструкция. Ако такава не бъде достигната, изпълнението ще продължи до достигане на края на **switch**.

От дадената дефиниция на конструкцията **switch** забелязваме, че след конструкцията на **default** липсва **break**. Това няма значение, защото в случая след **default** конструкцията няма други **case** конструкции за изпълнение, а се бележи края на тялото на **switch**. Програмистът може да постави след **default** конструкцията **break**, ако това е важно за добрия стил или за потока на изчисленията. Трябва да се има предвид, че не е задължително **default** конструкцията да е на последно място, тя може да бъде поставена някъде във вътрешността на switch конструкцията.

Правила за израза в switch

Конструкцията **switch** е един ясен начин за имплементиране на избор между множество варианти (тоест, избор между няколко различни пътища за изпълнение). Тя изисква селектор, който се изчислява до цяло число от типа **int**, **byte**, **char** или **enum**. Ако искаме да използваме, например, низ или число с плаваща запетая като селектор, това няма да работи в **switch** конструкция. За нецелочислени типове данни трябва да използваме последователност от **if** конструкции.

Използване на множество етикети

Използването на множество етикети е удачно, когато искаме да бъде изпълнена една и съща конструкция в повече от един случай. Нека разгледаме следния пример:

```
int number = 6;
switch (number) {
   case 1:
   case 4:
   case 6:
   case 8:
   case 10: System.out.println("Числото не е просто!"); break;
   case 2:
   case 3:
   case 5:
   case 7: System.out.println("Числото е просто!"); break;
   default: System.out.println("Не знам какво е това число!");
```

```
}
```

В този пример е имплементирано използването на множество етикети чрез написването на case конструкции без **break** след тях, така че в случая първо ще се изчисли целочислената стойност на селектора – тук тя е 6, и след това тази стойност ще започне да се сравнява с всяка една целочислена стойност в **case** конструкциите. Въпреки, че ще бъде срещнато съвпадение със стойността на селектора и третата case конструкция, изпълнението ще продължи надолу до срещането на първия **break**.

Резултатът от предходния програмен код е:

```
Числото не е просто
```

Добри практики при използване на switch-case

- Добра практика при използването на конструкцията за избор на вариант **switch** е **default** конструкцията да бъде поставяна на последно място, с цел програмния код да бъде по-лесно четим.
- Важно е да се използват отделни case конструкции за обработка на различни ситуации.
- Добре е на първо място да бъдат поставяни онези case конструкции, които обработват най-често възникващите ситуации. **case** конструкциите, които обработват ситуации, възникващи по-рядко могат да бъдат поставени на последно място.
- Добре е case конструкциите да бъдат подреждани в азбучен ред, ако целочислените стойности, с които се сравнява селекторът, са от символен тип.
- Добре е **case** конструкциите да бъдат подреждани в нарастващ ред, ако целочислените стойности, с които се сравнява селекторът, са от целочислен тип.
- Добре е да се използва **default** конструкция за ситуации, които не могат да бъдат обработени при нормално изпълнение на програмата.

Упражнения

1. Да се напише **if**-конструкция, която изчислява стойността на две целочислени променливи и разменя техните стойности, ако стойността на първата променлива е по-голяма от втората.
2. Напишете програма, която показва знака (**+** или **-**) от частното на две реални числа, без да го пресмята.
3. Напишете програма, която намира най-голямото по стойност число, измежду три дадени числа.

4. Напишете програма, която за дадена цифра (0-9), зададена като вход, извежда името на цифрата на български език.
5. Напишете програма, която при въвеждане на коефициентите (**a**, **b** и **c**) на квадратно уравнение: ax^2+bx+c, изчислява и извежда неговите реални корени.
6. Напишете програма, която намира най-голямото по стойност число измежду дадени 5 числа.
7. Дадени са няколко цели числа. Напишете програма, която намира онези подмножества от тях, които имат сума 0. Примери:
   - Ако са дадени числата {-2, -1, 1}, сумата на -1 и 1 е 0.
   - Ако са дадени числата {3, 1, -7}, няма подмножества със сума 0.
8. Напишете програма, която прилага бонус точки към дадени точки в интервала [1..9] чрез прилагане на следните правила:
   - Ако точките са между 1 и 3, програмата ги умножава по 10.
   - Ако точките са между 4 и 6, ги умножава по 100.
   - Ако точките са между 7 и 9, ги умножава по 1000.
   - Ако точките са 0 или повече от 9, се отпечатва съобщение за грешка.
9. Напишете програма, която преобразува дадено число в интервала [0..999] в текст, съответстващ на българското произношение. Примери:
   - 0 $\rightarrow$ "Нула"
   - 273 $\rightarrow$ "Двеста седемдесет и три"
   - 400 $\rightarrow$ "Четиристотин"
   - 501 $\rightarrow$ "Петстотин и едно"
   - 711 $\rightarrow$ "Седемстотин и единадесет"

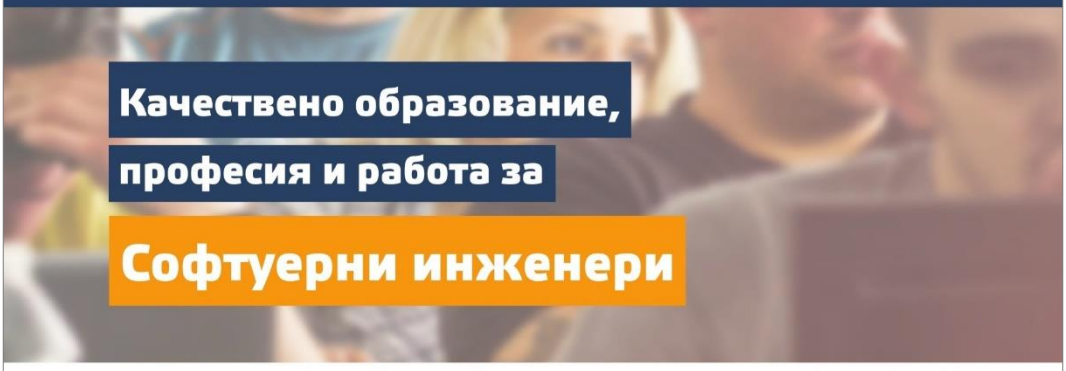
Решения и упътвания

1. Погледнете секцията за if конструкции.
2. Трябва да използвате последователност от **if** конструкции.
3. Трябва да използвате вложени **if** конструкции.
4. Трябва да използвате **switch** конструкция.
5. От математиката е известно, че едно квадратно уравнение може да има един или два реални корена или въобще да няма реални корени. За изчисляване на реалните корени на дадено квадратно уравнение първо се намира стойността на дискриминантата (D) по следната формула: $D=\sqrt{b^2-4ac}$. Ако стойността на дискриминантата е нула, то квадратното уравнение има един двоен реален корен. Той се изчислява по следната формула: $x_{1,2}=\frac{-b}{2a}$. Ако стойността на дискриминантата е

положително число, то уравнението има два различни реални корени, които се изчисляват по следната формула: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Ако стойността на дискриминантата е отрицателно число, то квадратното уравнение няма реални корени.

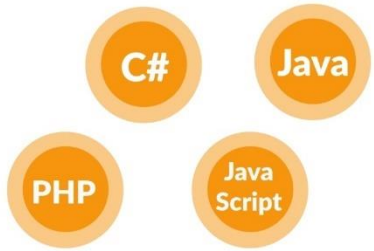
6. Използвайте вложени **if** конструкции. Можете да използвате конструкцията за цикъл **for**, за която можете да прочетете в Интернет.
7. Използвайте вложени **if** конструкции.
8. Използвайте **switch** конструкция и накрая изведете като резултат на конзолата пресметнатата точка.
9. Използвайте вложени **switch** конструкции. Да се обърне внимание на числата от 0 до 19 и на онези, чиито единици е нула.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 6. Цикли

Автор

Румяна Топалска

В тази тема...

В настоящата тема ще разгледаме конструкциите за цикли, с които можем да изпълняваме даден фрагмент програмен код многократно. Ще разгледаме как се реализират повторения с условие (**`while`** и **`do-while`** цикли) и как се работи с **`for`**-цикли. Ще дадем примери за различните възможности за дефиниране на цикъл, за начина им на конструиране и за някои от основните им приложения. Накрая ще разгледаме как можем да използваме няколко цикъла един в друг (вложени цикли).

Какво е "цикъл"?

В програмирането често се налага многократното изпълнение на дадена последователност от операции. **Цикълът (loop)** е структурата, която позволява това изпълнение без излишно писане на повтарящ се код. В зависимост от вида на цикъла, програмния код в него се повтаря:

- определени от фиксирано число пъти;
- докато дадено условие е изпълнено.

Цикъл, който никога не свършва, се нарича **безкраен (infinite)**.

Конструкция за цикъл while

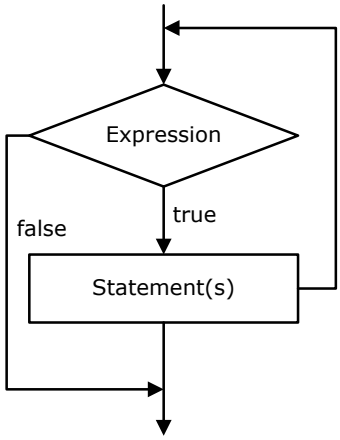
Един от най-простите и често използвани цикли е **while**.

```
while (condition) {
  statements;
}
```

Condition е израз, който връща булев резултат – **`true`** или **`fasle`**. Той определя докога ще се изпълнява тялото на цикъла и се нарича – **loop condition.**

Statements са програмният код, изпълняван в цикъла. Те се наричат тяло на цикъла.

При **while** цикъла първо се изпълнява булевия израз, ако резултатът от него е true, се изпълнява и последователността от операции, това се повтаря докато условния израз не върне **false**. Това е и причината често да бъде наричан цикъл с предусловие (**pre-test loop**). Ето логическата схема, по която се изпълняват **while** циклите:



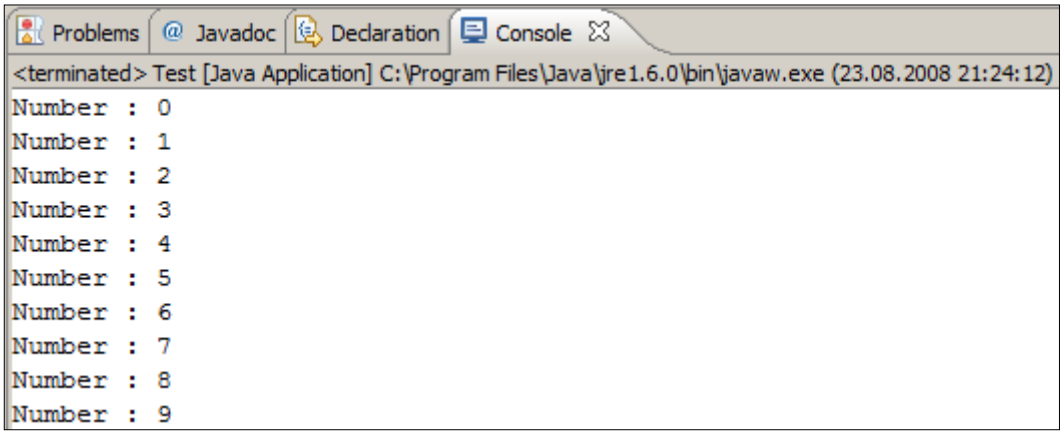
Използване на while цикли

While циклите изпълняват група операции докато е в сила дадено условие.

Нека разгледаме един съвсем прост пример за използването на **while** цикъл, при който само се отпечатват на конзолата числата в интервала от 0 до 9 в нарастващ ред:

```
// Initialize the counter
int counter = 0;
// Check the loop condition
while (counter < 10) {
   // Execute statements in loop if the result is true
   System.out.printf("Number : %d%n", counter);
   // Change the counter
   counter++;
}
```

При изпълнение на примерния код получаваме следния резултат:



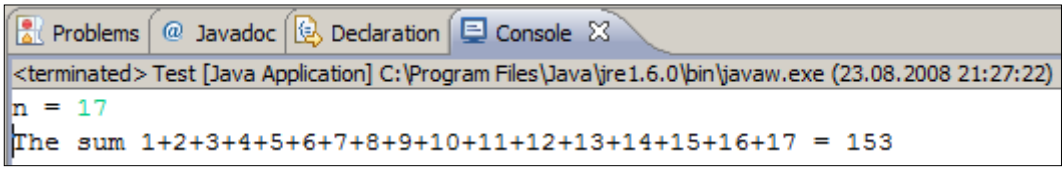
Нека дадем още примери, за да се убедите в ползата от циклите и да ви покажем какви задачи можем да решаваме с цикли.

Сумиране на числата от 1 до N – пример

В настоящия пример ще разгледаме как с помощта на цикъла **while** се намира сумата на числата от **1** до **N**. Числото **N** се чете от конзолата. Инициализираме променливите **num** и **sum** със стойност **1**. В **num** ще пазим текущото число, което ще добавяме към сумата на предходните. При всяко преминаване през цикъла ще го увеличаваме с **1**, за да получим следващото число, след което при влизане в цикъла проверяваме дали то отговаря на условието на цикъла, тоест дали е в интервала от **1** до **N**. **Sum** е променливата за сумата на числата. При влизане в цикъла добавяме към нея поредното число записано в **num**. На конзолата принтираме всички **num** (числа от 1 до N) с разделител "+" и крайния резултат от сумирането след приключване на цикъла.

```
Scanner input = new Scanner(System.in);
System.out.print("n = ");
int n = input.nextInt();
int num = 1;
int sum = 1;
System.out.print("The sum 1");
while (num < n) {
   num++;
   sum += num;
   System.out.printf("+%d", num);
}
System.out.printf(" = %d%n", sum);
```

Изходът е програмата е следният:



Нека дадем още един пример за използване на **while**, преди да продължим към другите конструкции за организиране на цикъл.

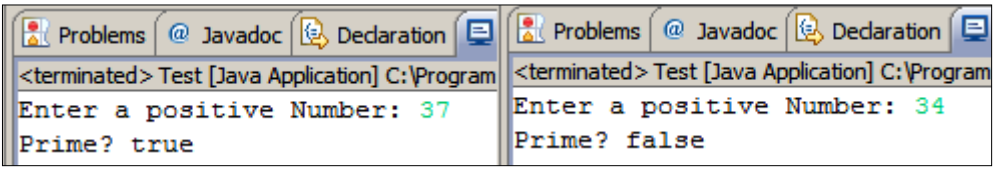
Проверка за просто число – пример

Ще напишем програма, с която да проверяваме дали дадено число е просто. Числото ще четем от конзолата. Както знаем от математиката, просто е всяко число, което освен на себе си и на 1, не се дели на други числа. Можем да проверим дали числото **num** е просто, като в цикъл проверим дали се дели на всички числа между 2 и √**num**:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a positive Number: ");
int num = input.nextInt();
int divider = 2;
int maxDivider = (int) Math.sqrt(num);
boolean prime = true;
while (prime && (divider <= maxDivider)) {
   if (num % divider == 0) {
      prime = false;
   }
   divider++;
}
System.out.println("Prime? " + prime);
```

Променливата **divider**, използваме за стойността на евентуалния делител на числото, я инициализираме в началото с 2. **maxDivider** е максималният възможен делител, който е равен на корен квадратен от числото (ако имаме делител, по-голям от √**num**, то би трябвало **num** да има и друг делител, който е обаче по-малък от √**num** и затова няма смисъл да проверяваме числата, по-големи от √**num**). Така намаляваме броя на итерациите на цикъла. Друга променлива от булев тип **prime** ще използваме за резултата. Първоначално, нека нейната стойност е **true**. При преминаване през цикъла, ако се окаже, че числото има делител, стойността ще стане **false**. В този пример условието на **while** цикъла се състои от две подусловия, които са свързани с логическия оператор **&&** (логическо и). В този случай, за да бъде изпълнен цикъла, трябва и двете да са верни едновременно, което също води до намаляване броя на итерациите, защото намирайки един делител **prime** става **false** и условието вече не е изпълнено. Това означава, че цикъла се изпълнява до намиране на първия делител на числото.

Ето как изглежда резултатът от изпълнението на горния пример:



Оператор break

Операторът **break** се използва за излизане от цикъла. Операциите в цикъла се изпълняват в съответния им ред и при достигане на оператора **break**, независимо дали условието за излизане от цикъла е изпълнено, изпълнението на цикъла се прекратява, като кода след **break** не се изпълнява.

Изчисляване на факториел – пример

В този пример ще пресметнем факториела на въведено през конзолата число с помощта на безкраен **while** цикъл и оператора **break**. Да си припомним от математиката какво е факториел и как се изчислява. Това е функция на естествено число n, която изразява произведението на всички естествени числа, по-малки или равни на n. Записва се n! и по дефиниция:

- n! = 1*2*3.....(n-1)*n, за n>1;
- 1! = 1;
- 0! = 1.

N! може да се изрази чрез факториел от естествени числа, по-малки от n:

- n! = (n-1)!n, като използваме началната стойност 1! = 1.

Това ще използваме и ние, за да изчислим факториела на n. Инициализираме променливата **factorial** с 1, а n – четем от конзолата. **While** цикълът, който ще конструираме, искаме да е безкраен. За тази цел условието трябва винаги да е **true**. Ще използваме оператора **break**, за да прекратим цикълът, когато n достигне 1. Самото изчисление ще започнем от числото **n**, с него ще умножаваме **factorial**, след което **n** ще намаляваме с 1. Или ще сметнем факториел по следната формула: n*(n-1)*(n-2)*…*3*2, при n=1 прекратяваме изпълнението на цикъла.

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
// "long" is the biggest integer type
long factorial = 1;
// Perform an “infinite loop"
while (true) {
   if (n == 1) {
      break;
   }
```

```
   factorial *= n;
   n--;
}
System.out.println("n! = " + factorial);
```

Ако въведем 10 като вход, на конзолата ще видим следния резултат:

```
10
n! = 3628800
```

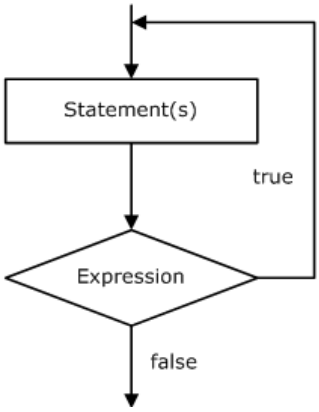
Конструкция за цикъл do-while

Do-while цикълът е аналогичен на **while** цикъла, само че при него проверката на булевия израз се прави след изпълнението на операциите в цикъла. Този тип цикли се наричат – цикли с условие в края (post-test loop).

Ето как изглежда един **do-while** цикъл:

```
do {
  statements;
}
while (expression);
```

Схематично **do-while** циклите се изпълняват по следната логическа схема:



Използване на do-while цикли

Do-while цикълът се използва, когато искаме да си гарантираме, че поредицата от операции в него ще бъде многократно, но май-малко веднъж.

Изчисляване на факториел – пример

В този пример отново ще изчислим факториела на дадено число **n**, но този път вместо безкраен **while** цикъл, ще използваме **do-while**. Логиката е

аналогична на тази в предния пример. Умножаваме всяко следващо число с произведението на предходните и го намаляваме с 1, след което проверяваме дали то все още е по-голямо от 0. Накрая отпечатваме получения резултат на конзолата.

```
Scanner input = new Scanner(System.in);
System.out.print("n = ");
int n = input.nextInt();
long factorial = 1;
do {
   // Multiply to become next value of factorial
   factorial *= n;
   // Decrement n to get next number
   n--;
} while (n > 0); // Check the loop condition
System.out.println("n! = " + factorial);
```

Ето го и резултатът от изпълнение на горния пример при n=7:

```
n = 7
n! = 5040
```

Нека дадем още един, по-интересен, пример.

Произведение в интервала [N...M] – пример

В този пример ще намерим произведението на всички числа в интервала [N…M]. Началната стойност на интервала я присвояваме на целочислената променлива **num**. Влизайки в цикъла първо ще пресмятаме произведението на числата до **num**. Това произведение ще записваме в **product**, след което **num** ще го увеличаваме с 1 и цикълът ще се изпълнява докато **num** не стане равно на **m**.

```
Scanner input = new Scanner(System.in);
// Read the end values of the interval n...m from the console
int n = input.nextInt();
int m = input.nextInt();
int num = n;
long product = 1;
do {
   product *= num;
   num++;
} while (num <= m);
System.out.println("product[n..m] = " + product);
```

Като резултат ще получим нещо такова:

```
2
6
product[n..m] = 720
```

Конструкция за цикъл for

For-циклите са малко по-сложни от **while** и **do-while** циклите, но за сметка на това могат да решават по-сложно задачи с по-малко писане на код. Характерната за **for**-цикъла структура е следната:

```
for (initialization; test; update) {
   statements;
}
```

Тя се състои от инициализационна част за брояча, булево условие, израз за обновяване на брояча и тяло на цикъла. Броячът на **for** цикъла го отличава от другите цикли. Броят на итерациите на този цикъл най-често се знае още преди да започне изпълнението му.

Безкраен цикъл (infinite loop) чрез оператора **for** се конструира по следния начин:

```
for ( ; ; ) {
   statements;
}
```

Безкраен цикъл означава цикъл, който никога не завършва. Обикновено в нашите програми нямаме нужда от безкраен цикъл, освен, ако някъде в тялото му не използваме **break**, за да завършим цикъла преждевременно.

Инициализация на for цикъла

For-циклите могат да имат инициализационен блок:

```
for (int num = 0; ...; ...) {
    // Can use num here
}
// Cannot use num here
```

Той се изпълнява веднъж, точно преди влизане в цикъла. Обикновено се използва за деклариране на променливата-брояч. Тази променлива е "видима" и може да се използва само в рамките на цикъла.

Условие на for цикъла

For-циклите могат да имат условие за повторение:

```
for (int num = 0; num < 10; ...) {
    // Can use num here
}
// Cannot use num here
```

То се изпълнява веднъж, преди всяка итерация на цикъла. При резултат **true** се изпълнява тялото на цикъла, а при **false** то се пропуска и се преминава към останалата част от програмата. Използва се като **loop condition** (условие на цикъла).

Обновяване на водещата променлива

Последната част от един for-цикъл съдържа код, който обновява водещата променлива:

```
for (int num = 0; num < 10; num++) {
    // Can use num here
}
// Cannot use num here
```

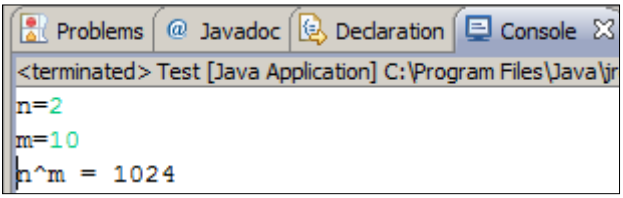
Той се изпълнява след като е приключило изпълнението на тялото на цикъла. Най-често се използва за обновяване стойността на брояча.

Изчисляване на N^M – пример

Ще напишем програма, която пресмята m-тата степен на число n. Ще използваме **for**-цикъл. Инициализираме променливата-брояч (**int i = 0**). Определяме условието на цикъла – **i < m**, така цикълът се изпълнява от **0** до **m-1** или точно **m** пъти. При всяко изпълнение на цикъла **n** ще се вдига на поредната степен и накрая ще принтираме резултата, за да видим правилно ли работи програмата.

```
Scanner input = new Scanner(System.in);
System.out.print("n=");
int n = input.nextInt();
System.out.print("m=");
int m = input.nextInt();
long result = 1;
for (int i = 0; i < m; i++) {
   result *= n;
}
System.out.println("n^m = " + result);
```

Ето как изглежда изходът от програмата при n=2 и m=10:



For-цикъл с няколко променливи

С конструкцията за **for**-цикъл можем да ползваме едновременно няколко променливи. Ето един пример, в който имаме два брояча. Единият се движи от 1 нагоре, а другият се движи от 10 надолу:

```
for (int small=1, large=10; small<large; small++, large--) {
   System.out.printf("%d %d\n", small, large);
}
```

Условието за прекратяване на цикъла е застъпване на броячите. В крайна сметка се получава следния резултат:

```
1 10
2 9
3 8
4 7
5 6
```

Оператор continue

Операторът **continue** спира текущата итерация на най-вътрешния цикъл, но не излиза от него. С помощта на следващия пример ще разгледаме как точно се използва този оператор.

Ще намерим сумата на всички нечетни естествени числа в интервала [1...n], който не се делят на 7. Ще използваме **for**-цикъл. При инициализиране на променливата ще и зададем стойност 1, тъй като това е първото нечетно естествено число в интервала [1...n]. Ще проверяваме дали **i** е все още в интервала (**i <= n**). В израза за обновяване на променливата ще я увеличаваме с 2, за да работим само с нечетни числа. В тялото на цикъла ще правим проверка дали числото се дели на 7. Ако това е изпълнено извикваме оператора **continue**, който ще прекрати изпълнението на цикъла (няма да добави текущото число към сумата). Ще се извърши обновяване на променливата и ще продължи изпълнението на цикъла. Ако не е изпълнено ще премине към обновяване на сумата с числото.

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
```

```
int sum = 0;
for (int i = 1; i <= n; i += 2) {
   if (i % 7 == 0) {
      continue;
   }
   sum += i;
}
System.out.println("sum = " + sum);
```

Резултатът от примера при n=26 е следният:

```
26
sum = 141
```

Разширена конструкция за цикъл for

От версия 5 на Java за удобство на програмистите беше добавена още една конструкция за цикъл, т.нар. **foreach** конструкция, наричана още разширен **for**-цикъл. Тази конструкция служи за обхождане на всички елементи на даден масив, списък или колекция от елементи. Подробно с масивите ще се запознаем в темата "Масиви", но за момента можем да си представяме един масив като последователност от числа или други елементи.

Ето как изглежда един разширен **for**-цикъл:

```
for (variable : collection) {
   statements;
}
```

Както виждате, той е значително по-прост от стандартния **for**-цикъл и затова много-често се предпочита от програмистите, тъй като спестява писане.

Ето един пример, в който ще видите разширения **for**-цикъл в действие:

```
int[] numbers = {2, 3, 5, 7, 11, 13, 17, 19};
for (int i : numbers) {
   System.out.printf("%d ", i);
}
System.out.println();

String[] towns = {"Sofia", "Plovdiv", "Varna", "Bourgas"};
for (String town: towns) {
   System.out.printf("%s ", town);
}
```

В примера се създава масив от числа и след това те се обхождат с разширения for-цикъл и се отпечатват на конзолата. След това се създава

масив от имена на градове (символни низове) и по същия начин се отпечатват на конзолата. Ето какъв е резултатът от примера:

```
2 3 5 7 11 13 17 19
Sofia Plovdiv Varna Bourgas
```

Вложени цикли

Вложените цикли представляват конструкция от няколко цикъла един в друг. Най-вътрешния цикъл се изпълнява най-много пъти. В примерната конструкция по долу ще разгледаме пример за вложен цикъл. След инициализация на първия **for** цикъл ще започне да се изпълнява втория. Ще се инициализира променливата му, ще се провери условието и ще се изпълнят изразите в него, след което ще се обнови променливата и ще продължи изпълнението на този цикъл, докато условието му не върне **false**. В този случай ще се върне в първия **for** цикъл, ще се извърши обновяване на неговата променлива и отново ще бъде изпълнен целия втори цикъл. Обикновено 2 **for** цикъла се използват за манипулация на двумерни масиви. Вложените цикли, използвани необмислено, могат да влошат производителността на програмата.

```
for (initialization; test; update) {
   for (initialization; test; update) {
      statements;
   }
   …
}
```

Отпечатване на триъгълник – пример

Нека си поставим следната задача: по дадено число **n** да отпечатаме на конзолата триъгълник с **n** на брой реда, изглеждащ по следния начин:

```
1
1 2
1 2 3
. . .
1 2 3 . . . n
```

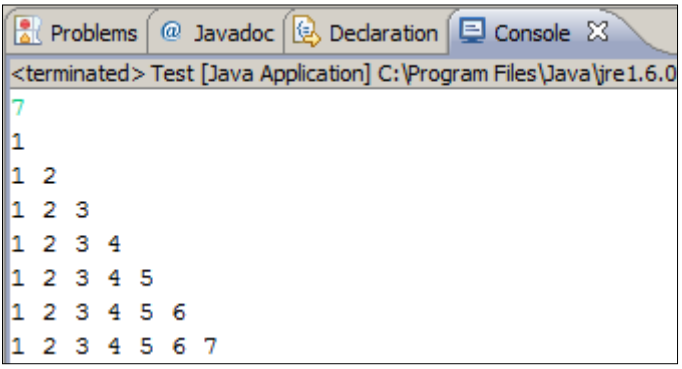
Това ще направим с два **for**-цикъла. Външния ще ни обхожда редовете, а вътрешния – елементите в тях. Когато сме на първия ред, трябва да отпечатаме "1" (1 елемент, 1 итерация на вътрешния цикъл). На втория – "1 2" (2 елемента, 2 итерации). Виждаме, че има зависимост между реда, на който сме и броя на елементите, който ще отпечатваме. Това ни дава информация за определяне конструкцията на вътрешния цикъл:

- инициализираме променливата с 1 (първото число, което ще отпечатаме) => **col** = 1;
- условието ни зависи от реда, на който сме. Той ограничава елементите => **col** <= **row**;
- обновяваме променливата увеличавайки я с 1.

На практика трябва да направим един **for**-цикъл (външен) от 1 до **n** (за редовете) и в него още един **for**-цикъл (вътрешен) за числата в текущия ред, който да е от 1 до номера на текущия ред. Външният цикъл ходи по редовете, а вътрешният – по всяка от колоните за текущия ред. В крайна сметка получаваме следния сорс код:

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
for (int row = 1; row <= n; row++) {
   for (int col = 1; col <= row; col++) {
      System.out.print(col + " ");
   }
   System.out.println();
}
```

Ако го изпълним, ще се убедим, че работи коректно. Ето как изглежда резултатът при n=7:



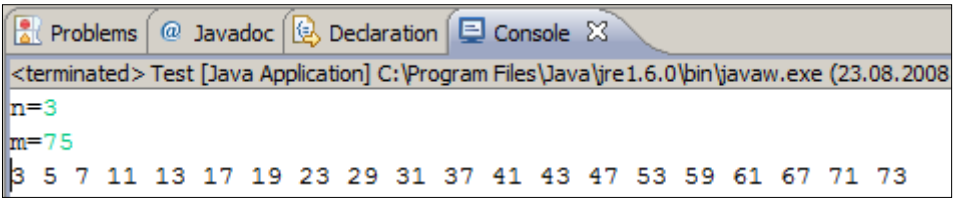
Прости числа в даден интервал – пример

Да разгледаме още един пример за вложени цикли. Поставяме си за цел да отпечатаме на конзолата всички прости числа в интервала [N, M]. Интервалът ще ограничим с **for**-цикъл, а за проверката за просто число ще използваме вложен **while** цикъл. Логиката, по която правим проверката за просто число, вече ни е позната. Във **for**-цикъла инициализираме променливата да е равна на началната стойност на интервала, ще проверяваме при всяка итерация дали нейната стойност все още е в него и накрая ще увеличаваме с 1. След преминаване през **while** цикъла булевата променлива **prime** показва дали числото е просто или не. Това ще проверим

с условния оператор **if** и при резултат **true** ще отпечатваме числото на конзолата. Ето как изглежда реализацията:

```
Scanner input = new Scanner(System.in);
System.out.print("n=");
int n = input.nextInt();
System.out.print("m=");
int m = input.nextInt();
for (int num = n; num <= m; num++) {
   boolean prime = true;
   int divider = 2;
   int maxDivider = (int) Math.sqrt(num);
   while (divider <= maxDivider) {
      if (num % divider == 0) {
         prime = false;
         break;
      }
      divider++;
   }
   if (prime) {
      System.out.printf("%d ", num);
   }
}
```

Ако изпълним примера за n=3 и m=75 ще получим следния резултат:



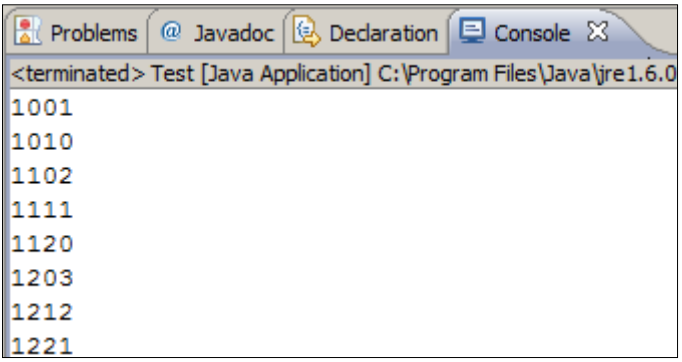
Щастливи числа – пример

Нека разгледаме още един пример, с който ще покажем, че можем да влагаме и повече от два цикъла един в друг. Целта е да намерим и отпечатаме всички четирицифрени числа от вида ABCD, за които: A+B = C+D (наричаме ги щастливи числа). Това ще реализираме с помощта на четири **for**-цикъла – за всяка цифра по един. Най-външния цикъл ще ни определя хилядните. Той ще започва от 1, а останалите от 0. Ще правим проверка, дали текущото ни число е щастливо, в най-вътрешния цикъл. Ако е така ще го отпечатваме на конзолата.

```
for (int a = 1; a <= 9; a++) {
   for (int b = 0; b <= 9; b++) {
      for (int c = 0; c <= 9; c++) {
         for (int d = 0; d <= 9; d++) {
```

```
            if ((a + b) == (c + d)) {
               System.out.printf("%d%d%d%d%n", a, b, c, d);
            }
         }
      }
   }
}
```

Ето част от отпечатания резултат (целият е много дълъг):



ТОТО 6/49 – пример

В този пример ще намерим всички възможни комбинации от тотото (6/49). Трябва да намерим и отпечатаме всички възможни извадки от 6 различни числа в интервала [1...49]. Ще използваме 6 **for** цикъла. За разлика от предния пример, има изискване, че числата не могат да се повтарят. Затова вътрешните цикли няма да започват от 1, а от числото, до което е стигнал предходния цикъл + 1. За да избегнем повторенията ще се стремим всяко следващо число да е по-голямо от предходното. Първият цикъл ще трябва да го въртим до 44 (а не до 49), вторият до 45, и т.н. Последният цикъл ще е до 49. Ако въртим всички цикли до 49, ще получим съвпадащи числа в някои от комбинациите. По същата причина всеки следващ цикъл започва от брояча на предходния + 1. Нека да видим какво ще се получи:

```
for (int i1 = 1; i1 <= 44; i1++)
   for (int i2 = i1 + 1; i2 <= 45; i2++)
      for (int i3 = i2 + 1; i3 <= 46; i3++)
         for (int i4 = i3 + 1; i4 <= 47; i4++)
            for (int i5 = i4 + 1; i5 <= 48; i5++)
               for (int i6 = i5 + 1; i6 <= 49; i6++)
                  System.out.printf(
                     "%d %d %d %d %d %d%n",
                     i1, i2, i3, i4, i5, i6);
```

Всичко изглежда правилно. Да стартираме програмата. Изглежда, че работи, но има един проблем – комбинациите са прекален много и програмата не завършва (едва ли някой ще я изчака). Това е в реда на нещата и е една от причините да има ТОТО 6/49 – комбинациите наистина са много.

Упражнения

1. Напишете програма, която отпечатва на конзолата числата от 1 до N. Числото N се чете от стандартния вход.
2. Напишете програма, която отпечатва на конзолата числата от 1 до N, които не се делят на 3 и 7. Числото N се чете от стандартния вход.
3. Напишете програма, която чете от конзолата поредица от цели числа и отпечатва най-малкото и най-голямото от тях.
4. Напишете програма, която отпечатва всички възможни карти от стандартно тесте без джокери (имаме 52 карти: 4 бои по 13 карти).
5. Напишете програма, която чете от конзолата числото N и отпечатва сумата на първите N члена от редицата на Фибоначи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
6. Напишете програма, която пресмята N!/K! за дадени N и K (1<K<N).
7. Напишете програма, която пресмята N!*K!/(N-K)! за дадени N и K.
8. Напишете програма, която за дадено цяло число N, пресмята сумата: $S = 1 + {1!}/{x} + {2!}/{x^2} + ... + {n!}/{x^n}$
9. В комбинаториката числата на Каталан (Catalan's numbers) се изчисляват по следната формула: $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$, за n ≥ 0. Напишете програма, която изчислява N-тото число на Каталан за дадено N.
10. Напишете програма, която чете от конзолата положително цяло число N (N < 20) и отпечатва матрица с числа като на фигурата по-долу:

N = 3

1	2	3
2	3	4
3	4	5

N = 4

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

11. Напишете програма, която пресмята на колко нули завършва факториела на дадено число. Примери:

N = 10 -> N! = 3628800 -> 2

N = 20 -> N! = 2432902008176640000 -> 4

12. Напишете програма, която преобразува дадено число от десетична в двоична бройна система.
13. Напишете програма, която преобразува дадено число от двоична в десетична бройна система.
14. Напишете програма, която преобразува дадено число от десетична в шестнайсетична бройна система.
15. Напишете програма, която преобразува дадено число от шестнайсетична в десетична бройна система.
16. Напишете програма, която по дадено число N отпечатва случайно число между 1 и N.

Решения и упътвания

1. Използвайте **`for`** цикъл.
2. Използвайте **`for`** цикъл и оператора **`%`** за намиране на остатък при целочислено деление.
3. Първо прочетете броя числа. След това ги въведете с един **`for`** цикъл. Докато въвеждате всяко следващо число запазвайте в две променливи най-малкото и най-голямото число до момента.
4. Номерирайте картите от 2 до 14 (тези числа ще съответстват на картите от 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A). Номерирайте боите от 1 до 4 (1 - спатия, 2 – каро, 3 – купа, 4 – пика). Сега вече можете да завъртите 2 вложени цикъла и да отпечатате всяка от картите.
5. Числата на Фибоначи започват от 0 и 1, като всяко следващо се получава като сума от предходните две.
6. Умножете числата от K+1 до N.
7. Вариант за решение е поотделно да пресмятате всеки от факториелите и накрая да извършвате съответните операции с резултатите. Помислете как можете да оптимизирате пресмятанията, за да не смятате прекалено много факториели! При обикновени дроби, съставени от факториели има много възможности за съкращение на еднакви множители в числителя и знаменателя.
8. Погледнете предходната задача.
9. Погледнете предходната задача.
10. Трябва да използвате два вложени цикъла, по подобие на задачата с триъгълника.

11. Броят на нулите зависи от това, колко пъти числото 5 е делител на факториела.
12. Прочетете в Уикипедия какво представляват бройните системи: http://en.wikipedia.org/wiki/Numeral_system. След това помислете как можете да преминавате от десетична в друга бройна система. Помислете и за обратното – преминаване от друга бройна система към десетична. Ако се затрудните, вижте главата "Бройни системи".
13. Погледнете предходната задача.
14. Погледнете предходната задача.
15. Погледнете предходната задача.
16. Потърсете в Интернет информация за класа **`java.util.Random`**.

Глава 7. Масиви

Автор

Мариян Ненчев

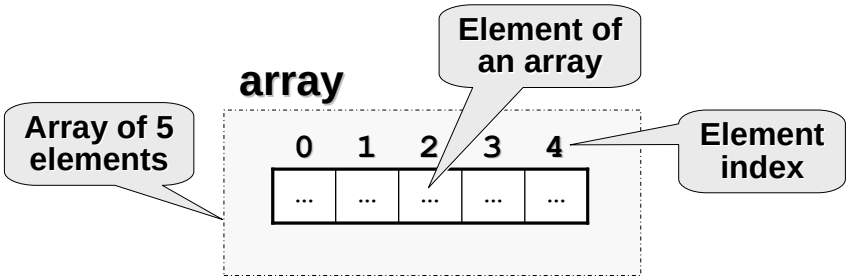
Светлин Наков

В тази тема...

В настоящата тема ще се запознаем с масивите като средства за обработка на поредица от еднакви по тип елементи. Ще обясним какво представляват масивите, как можем да декларираме, създаваме и инициализираме масиви. Ще обърнем внимание на едномерните и многомерните масиви. Ще разгледаме различни начини за обхождане на масив, четене от стандартния вход и отпечатване на стандартния изход. Ще дадем много примери за задачи, които се решават с използването на масиви и ще ви покажем колко полезни са те.

Какво е "масив"?

Масивите са неизменна част от езиците за програмиране. Те представляват съвкупности от променливи, които наричаме **елементи**:



Елементите на масивите са номерирани с числата 0, 1, 2, ... N-1. Тези номера на елементи се наричат **индекси**. Броят елементи в даден масив се нарича **дължина на масива**.

Всички елементи на даден масив са от един и същи тип, независимо дали е **примитивен** или **референтен**. Това ни помага да представим група от еднородни елементи като подредена свързана последователност и да ги обработваме като едно цяло.

Масивите могат да бъдат от различни размерности, като най-често използвани са **едномерните** и **двумерните** масиви. Едномерните масиви се наричат още вектори, а двумерните матрици.

Деклариране и заделяне на масиви

В Java масивите имат фиксирана дължина, която се указва при инициализирането му и определя броя на елементите му. След като веднъж сме задали дължината на масив не е възможно да я променяме.

Деклариране на масив

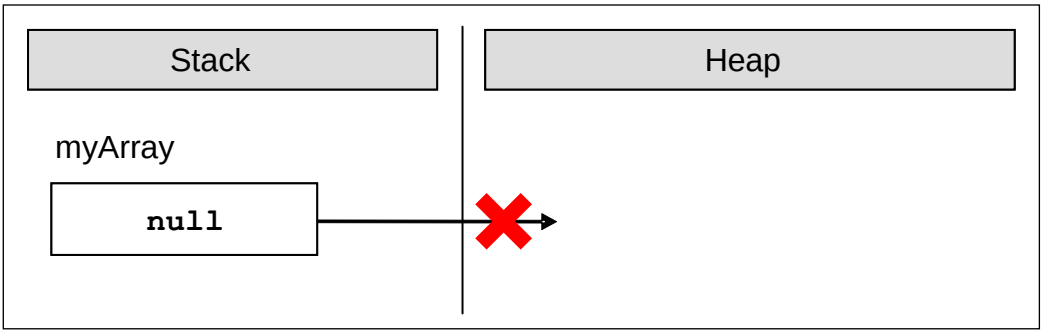
Масиви в Java декларираме по следния начин:

```
int[] myArray;
```

Тук променливата **myArray** е името на масива, който е от тип (**int[]**) т.е. декларирали сме масив от цели числа. С **[]** се обозначава, че променливата, която декларираме ще е масив, а не единичен елемент.

При декларация името на променливата, която е от тип масив, представлява референция (**reference**), която сочи към **null**, тъй като още не е заделена памет за елементите на масива.

Ето как изглежда една променлива от тип масив, която е декларирана, но още не е заделена памет за елементите на масива:



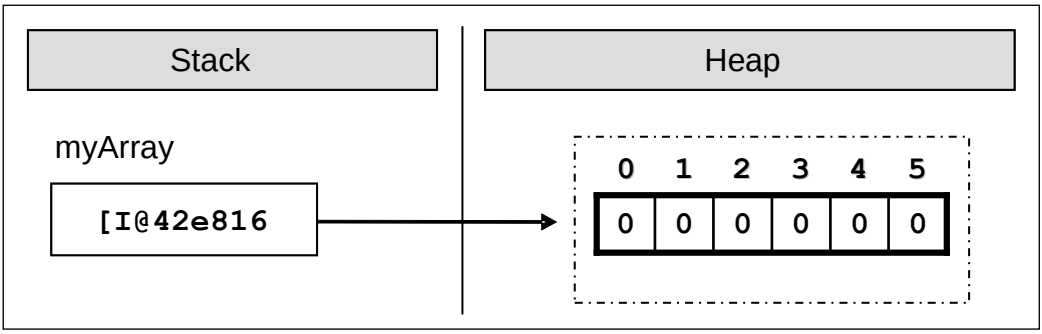
В стека за изпълнение на програмата се заделя променлива с име **myArray** и в нея се поставя стойност **null** (липса на стойност).

Създаване (заделяне) на масив – оператор new

В Java масив се създава с ключовата дума **new**, която служи за заделяне (алокиране) на памет:

```
int[] myArray = new int[6];
```

В примера заделяме масив с размер 6 елемента от целочислен тип. Това означава, че в динамичната памет (heap) се заделя участък от 6 последователни цели числа:



Картинката показва, че след заделянето на масива променливата **myArray** сочи някакъв адрес (**0x42e816**) в динамичната памет, където се намира нейната стойност. Елементите на масивите винаги се съхраняват в динамичната памет (в т. нар. **heap**).

При заделянето на масив в квадратните скоби задаваме броя на елементите му (цяло неотрицателно число) и така се фиксира неговата дължина. Типът на елементите се пише след **new**, за да се укаже за какви точно елементи трябва да се задели памет. Масив с вече зададена дължина не може да се промени т.е. масивите са с фиксирана дължина.

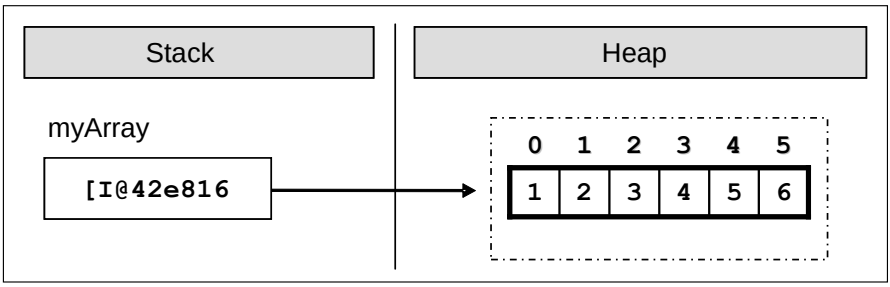
Инициализация на масив. Стойности по подразбиране

Преди да използваме елемент от даден масив той трябва да има начална стойност. В някои езици за програмиране не се задават начални стойности по подразбиране, и тогава при опит за достъпване да даден елемент възниква грешка. В Java всички променливи, включително и елементите на масивите имат начална стойност по подразбиране (default initial value)> Тази стройност е равна на 0 при числените типове или неин еквивалент при нечислени типове (например **null** за обекти и **false** за булевия тип).

Разбира се, начални стойности можем да задаване и изрично. Това може да стане по различни начини. Един възможен е чрез използване на литерален израз за елементите на масива (**array literal expression**):

```
int[] myArray = {1, 2, 3, 4, 5, 6};
```

В този случай създаваме и инициализираме масива едновременно. Ето как изглежда масива в паметта, след като стойностите му са инициализирани още в момента на деклариране:



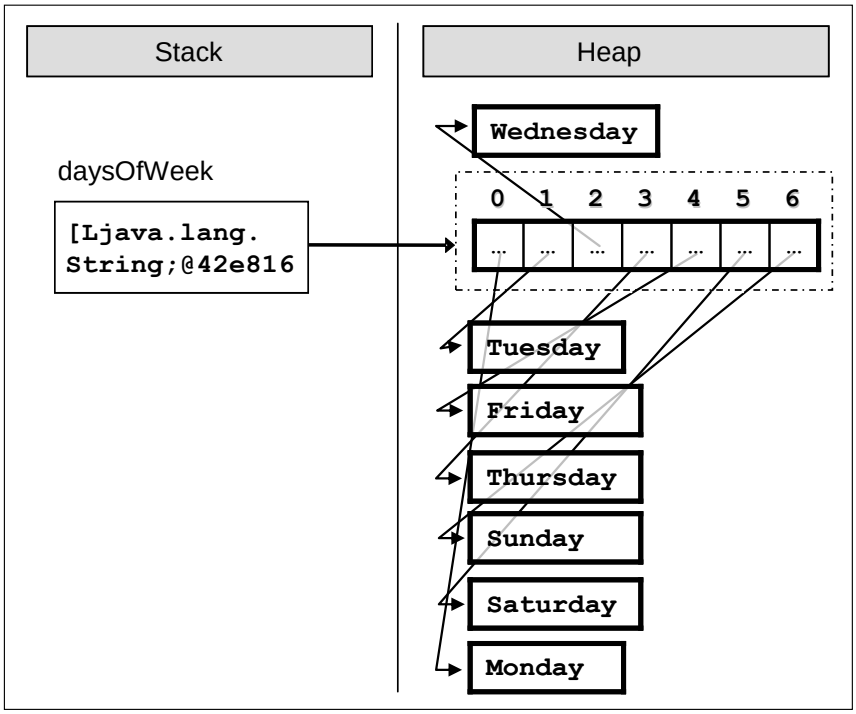
При този синтаксис къдравите скоби заместват оператора **new** и между тях има изброени началните стойности на масива, разделени със запетаи. Техния брой определя дължината му.

Деклариране и инициализиране на масив – пример

Ето още един пример за деклариране и непосредствено инициализиране на масив:

```
String[] daysOfWeek = {
   "Monday", "Tuesday", "Wednesday", "Thursday",
   "Friday", "Saturday", "Sunday" };
```

В случая масивът се заделя със 7 елемента от тип **String**. Типът String е референтен тип (обект) и неговите стойности се пазят в динамичната памет. Ето как изглежда масивът в паметта:



В стека се заделя променливата **daysOfWeek**, която сочи към участък в динамичната памет, който съдържа елементите на масива. Всеки от тези 7 елементи е обект от тип символен низ, който сам по себе си сочи към друга област от динамичната памет, в която се пази стойността му.

Достъп до елементите на масив

Достъпът до елементите на масивите е пряк, по индекс. Всеки елемент може да се достъпи с името на масива и съответния му индекс, поставен в квадратни скоби. Можем да осъществим достъп до даден елемент както за четене така и за писане т.е. да го третираме като обикновена променлива.

Масивите могат да се обхождат с помощта на някоя от структурите за **цикъл**, като най-често използван е класическият **for** цикъл.

Пример за достъп до елемент на масив:

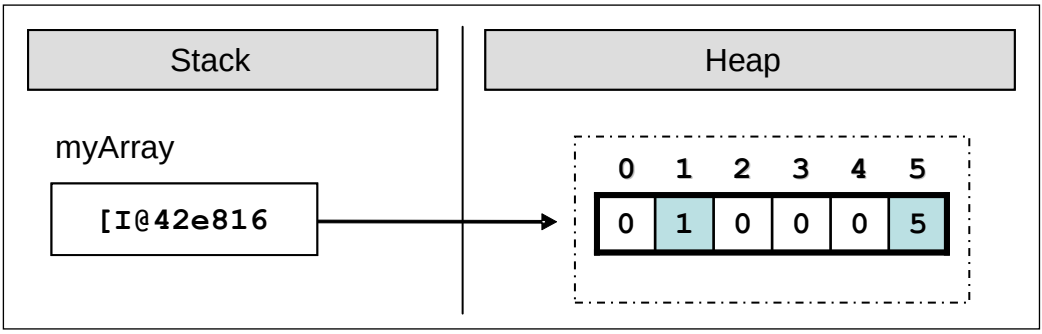
```
myArray[index] = 100;
```

В горния пример присвояваме стойност 100 на елемента, намиращ се на позиция **index**, където **index** е валиден за масива индекс.

Ето един пример, в който заделяме масив от числа и след това променяме някои от елементите му:

```
int[] myArray = new int[6];
myArray[1] = 1;
myArray[5] = 5;
```

След промяната на елементите, масивът се представя в паметта по следния начин:



Граници на масив

Масивите обикновено са **нулево-базирани**, т.е. номерацията на елементите започва от **0**. Първият елемент има индекс 0, вторият 1 и т.н. Ако един масив има **N** елемента, то последният елемент се намира на индекс **N-1**.

Излизане от границите на масив

Достъпът до елементите на масивите се проверява по време на изпълнение от виртуалната машина на Java и тя не допуска излизане извън границите и размерностите им. При всеки достъп до елемент на масива по се прави проверка, дали индексът е валиден или не. Ако не е се хвърля изключение от тип **java.lang.ArrayIndexOutOfBoundsException**. Естествено, тази проверка си има и своята цена и тя е леко намаляване на производителността.

Ето един пример, в който се опитваме да извлечен елемент, който се намира извън границите на масива:

TestArrayIndexOutOfBounds.java

```
public class TestArrayIndexOutOfBounds {
   public static void main(String[] args) {
      int[] myArray = { 1, 2, 3, 4, 5, 6 };
      System.out.println(myArray[6]);
   }
}
```

В горния пример създаваме масив, който съдържа 6 цели числа. Първият елемент се намира на индекс 0, последният на индекс 5. Опитваме се да изведем на конзолата елемент, който се намира на индекс 6, но такъв не съществува и това води до подаване на изключение:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
   at Tst.main(Tst.java:5)
```

Обръщане на масив в обратен ред – пример

В следващия пример ще видим как може да променяме елементите на даден масив като ги достъпваме по индекс. Ще обърнем елементите на масива, като използваме помощен масив, в който да запазим елементите на първия, но в обратен ред. Забележете, че дължината и на масивите е еднаква и остава непроменена след първоначалното им заделяне:

ArrayReverseExample.java

```
import java.util.Arrays;

public class ArrayReverseExample {
   public static void main(String[] args) {
      int[] array = new int[] { 1, 2, 3, 4, 5 };

      // Get array size
      int length = array.length;
```

```
        // Declare and create the reversed array
        int[] reversed = new int[length];

        // Initialize the reversed array
        for (int index = 0; index < length; index++) {
           reversed[length - index - 1] = array[index];
        }

        // Print the reversed array
        System.out.println(Arrays.toString(reversed));
    }
}
```

Ако изпълним примера, ще получим следния резултат:

```
[5, 4, 3, 2, 1]
```

Примерът работи така: първоначално създаваме едномерен масив от тип **int**, и го инициализираме с цифрите от 1 до 5. След това си запазваме дължината на масива в целочислената променлива **length**. Забележете, че се използва полето **length**, което връща броя на елементите на масива. В Java всеки масив знае своята дължина.

След това декларираме масив **reversed** с размер **length**, в който ще си пазим елементите на първия, но в обратен ред.

За да извършим обръщането на елементите ползваме цикъл **for**, като на всяка итерация увеличаваме **index** с единица и така си осигуряваме последователен достъп до всеки елемент на масива **array**. Критерия за край на цикъла ни подсигурява, че масива ще бъде обходен от край до край.

Нека проследим последователно какво се случва при итериране върху масива **array**. При първата итерация на цикъла, **index** има стойност 0. С **array[index]** достъпваме първия елемент на **array**, а с **reversed[length – index – 1]** достъпваме последния елемент на **reversed** и извършваме присвояване. Така на последния елемент на **reversed** присвоихме първия елемент на **array**. На всяка следваща итерация **index** се увеличава с единица, позицията в **array** се увеличава с единица, а в **reversed** се намаля с единица.

В резултат обърнахме масива в обратен ред. В примера показахме последователно обхождане на масив, което може да се извърши и с другите видове цикли.

Отпечатването на масив правим като се възползваме от помощния клас **java.util.Arrays** и метода му **toString()**, с който получаваме текстово представяне на масива.

Четене на масив от конзолата

Нека разгледаме как можем да прочетем стойностите на масив от конзолата. Ще използваме **for** цикъл и средствата на Java за четене на числа от конзолата.

Първоначално, за да заделим памет за масива, може да прочетем цяло число **n** от конзолата и да го ползваме като негов размер:

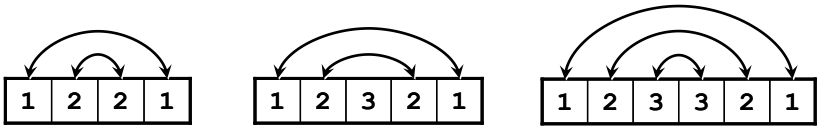
```
int n = input.nextInt();
int[] array = new int[n];
```

Отново използваме цикъл, за да обходим масива. На всяка итерация присвояваме на текущия елемент прочетеното от конзолата число. Цикъла ще се завърти **n** пъти т.е. ще обходи целия масив и така ще прочетем стойност за всеки елемент от масива:

```
for (int i = 0; i < n; i++) {
  array[i] = input.nextInt();
}
```

Проверка за симетрия на масив – пример

Един масив е симетричен, ако първият и последният му елемент са еднакви и вторият и предпоследният му елемент също са еднакви и т.н. На картинката са дадени няколко примера за симетрични масиви:



В следващия примерен код ще видим как може да проверим дали даден масив е симетричен:

```
int n = input.nextInt();
int[] array = new int[n];
for (int i = 0; i < n; i++) {
  array[i] = input.nextInt();
}
boolean symmetric = true;
for (int i = 0; i < (array.length + 1) / 2; i++) {
  if (array[i] != array[n - i - 1])
  symmetric = false;
}
System.out.printf("Symmetric? %b%n", symmetric);
```

Тук отново създаваме масив и прочитаме елементите му от конзолата. За да проверим дали масива е симетричен трябва да го обходим само до

средата му. Тя е равна на **(array.length + 1) / 2**, понеже не знаем дали масива има четна или нечетна дължина.

За да определим дали дадения масив е симетричен ще ползваме **булева** променлива, като по начало приемаме, че масивът е симетричен.

Обхождаме масива и сравняваме първия с последния елемент, втория с предпоследния и т.н. Ако за някоя итерация се окаже, че стойностите на елементите не съвпадат булевата променлива получава стойност **false**, т.е. масивът не е симетричен.

Най-накрая извеждаме на конзолата резултата на булевата променлива.

Отпечатване на масив на конзолата

Често се налага след като сме обработвали даден масив да изведем елементите му на конзолата, било то за тестови или други цели.

Отпечатването на елементите на масив става по подобен начин на инициализирането на елементите му, а именно като използваме цикъл, който обхожда масива. Няма строги правила за извеждането на данните. Разбира се, добра практика е те да бъдат добре форматирани.

Често срещана грешка е опит да се изведе на конзолата масив директно, по следния начин:

```
String[] array = { "one", "two", "three", "four" };
System.out.println(array);
```

Този код за съжаление не отпечатва съдържанието на масива, а неговия адрес в динамичната памет (защото масивите са референтни типове). Ето как изглежда резултатът от изпълнението на горния код:

```
[Ljava.lang.String;@42e816
```

За да изведем коректно елементите на масив на конзолата можем да използваме **for** цикъл:

```
String[] array = { "one", "two", "three", "four" };

// Process all elements of the array
for (int index = 0; index < array.length; index++) {
   // Print each element on a separate line
   System.out.printf("element[%d] = %s%n", index, array[index]);
}
```

Обхождаме масива с цикъл **for**, който извършва **array.length** на брой итерации, с помощта на метода **System.out.printf**, извеждаме данните на конзолата във **форматиран** вид. Резултатът е следният:

```
element[0] = one
element[1] = two
element[2] = three
element[3] = four
```

Има и още един, по-лесен начин да отпечатаме съдържанието на масив:

```
String[] array = { "one", "two", "three", "four" };
System.out.println(java.util.Arrays.toString(array));
```

Резултатът е добре форматиран символен низ, съдържащ всички елементи на масива, изброени със запетайка:

```
[one, two, three, four]
```

Итерация по елементите на масив

Както разбрахме до тук, итерирането по елементите на масив е една от основните операции при обработката на масиви. Итерирайки последователно по даден масив можем да достъпим всеки елемент с помощта на индекс и да го манипулираме по желан от нас начин. Това може да стане с всички видове конструкции за цикъл, които разгледахме в предните теми, но най-подходящ за това е стандартният **for** цикъл. Нека разгледаме как точно става обхождането на масив.

Итерация с for цикъл

Добра практика е да използваме **for** цикъл при работа с масиви и изобщо при индексирани структури. Ето един пример, в който удвояваме стойността на всички елементи от даден масив с числа:

```
int[] array = new int[] {1, 2, 3, 4, 5};
for (int index = 0; index < array.length; index++) {
   array[index] = 2 * array[index];
}
System.out.println(Arrays.toString(array));
// Output: [2, 4, 6, 8, 10]
```

Чрез **for** цикъла можем да имаме постоянен поглед върху текущия индекс на масива и да достъпваме точно тези елементи, от които имаме нужда. Итерирането може да не се извършва последователно т.е. индексът, който **for** цикъла ползва може да прескача по елементите според нуждите на нашия алгоритъм. Например можем да обходим част от даден масив, а не целия. Ето един пример:

```
for (int index = 0; index < array.length; index += 2) {
```

```
  array[index] = array[index] * array[index];
}
```

В горния пример обхождаме всички елементи на масива, намиращи се на четни позиции и повдигаме на квадрат стойността във всеки от тях.

Понякога е полезно да обходим масив отзад напред. Можем да постигнем това по напълно аналогичен начин, с разликата, че **for** цикълът ще започва с начален индекс, равен на индекса на последния елемент на масива, и ще се намаля на всяка итерация. Ето един такъв пример:

```
int[] array = new int[] {1, 2, 3, 4, 5};
System.out.print("Reversed: ");
for (int i = array.length - 1; i >= 0; i--) {
  System.out.print(array[i] + " ");
}
// Reversed: 5 4 3 2 1
```

В горния пример обхождаме масива от зад напред последователно и извеждаме всеки негов елемент на конзолата.

Итерация с разширен for цикъл (for-each)

Една често използвана вариация на **for** цикълът е така наречения **разширен for** цикъл, който наричаме **for-each**.

Конструкцията на **for-each** цикъла в Java е следната:

```
for (type value : array) {
   // Process the value here
}
```

При тази конструкция **type** е типът на елементите, които обхождаме т.е. типа на масива, **array** е масивът, а **value** е текущият елемент от масива на всяка една стъпка от обхождането.

For-each притежава свойствата на **for** цикъла. Отличава се с това, че обхождането на масива, въобще на структурата, която обхожда, се извършва от край до край. При него се скрива индекса на текущата позиция т.е. не знаем, на кой индекс се намира текущия елемент.

Този цикъл се използва когато не се нуждаем да променяме елементите на масива, а само да ги четем и да обхождаме целия масив.

For-each е по-бърз от обикновения **for** цикъл.

Итерация с разширен for цикъл (for-each) – пример

В следващия пример ще видим как да използваме конструкцията за разширен **for** цикъл за обхождане на масиви:

```
String[] capitals = {"Sofia", "Washington", "London", "Paris"};
for (String capital : capitals) {
   System.out.println(capital);
}
```

След като сме си декларирали масив от низове **capitals**, с **for-each** го обхождаме и извеждаме елементите му в конзолата. Текущия елемент на всяка една стъпка се пази в променливата **capital**. Ето какъв резултат се получава при изпълнението на примера:

```
Sofia
Washington
London
Paris
```

Многомерни масиви

До момента разгледахме работата с едномерни масиви, известни в математиката като "вектори". В практиката, обаче, често се ползват масиви с повече от едно измерения. Например стандартна шахматна дъска се представя лесно с двумерен масив с размер 8 на 8 (8 полета в хоризонтална посока и 8 полета във вертикална посока).

Какво е "многомерен масив"? Какво е "матрица"?

Всеки допустим в Java тип може да бъде използван за тип на елементите на масив. Масивите също може да се разглеждат като допустим тип. Така можем да имаме масив от масиви.

Едномерен масив от цели числа декларираме с **int[]**. Ако желаем да декларираме масив от масиви от тип **int[]**, трябва всеки елемент да е от тип **int[]**, т.е. получаваме декларацията:

```
int[][] twoDimentionalArray;
```

Такива масиви ще наричаме **двумерни**, защото имат две измерения или още **матрици** (терминът идва от математиката). Масиви с повече от едно измерение ще наричаме **многомерни**.

Аналогично можем да декларираме и **тримерни** масиви като добавим още едно измерение:

```
int[][][] threeDimentionalArray;
```

На теория няма ограничения за броя на размерностите на тип на масив, но в практиката масиви с повече от две размерности са рядко използвани, затова ще се спрем по-подробно на двумерните масиви.

Деклариране и заделяне на многомерен масив

Многомерните масиви се декларират по начин аналогичен на едномерните. Всяка размерност означаваме с квадратни скоби:

```
int[][] intMatrix;
float[][] floatMatrix;
String[][][] strCube;
```

Горният пример показва как да създадем двумерни и тримерни масиви. Всяка размерност отговаря на едни **[]**.

Памет за многомерни размери се заделя като се използва ключовата дума **new** и за всяка размерност в квадратни скоби се задава размера, който е необходим:

```
int[][] intMatrix = new int[3][4];
float[][] floatMatrix = new float[8][2];
String[][][] stringCube = new String[5][5][5];
```

В горния пример **intMatrix** е двумерен масив с 3 елемента от тип **int[]** и всеки от тези 3 елемента има размерност 4. Така представени, двумерните масиви изглеждат трудни за осмисляне. Затова може да ги разглеждаме като двумерни **матрици**, които имат редове и колони за размерности:

	0	1	2	3
0	1	2	6	3
1	9	0	7	1
3	2	8	5	4

Редовете и колоните се номерират с индекси от 0 до големината на съответната размерност минус едно. Ако един двумерен масив има размер **m** на **n**, той има **m*n** елемента.

Понякога можем да имаме неправоъгълни двумерни масиви, в които на всеки ред има различен брой колони.

Инициализация на многомерен масив

Инициализацията на многомерни масиви е аналогична на инициализацията на едномерните. Стойностите на елементите могат да се изброяват непосредствено след декларацията:

```
int[][] matrix = {
   {1, 2, 3, 4}, // row 0 values
   {5, 6, 7, 8}, // row 1 values
};
```

```
// The matrix size is 2 x 4 (2 rows, 4 cols)
```

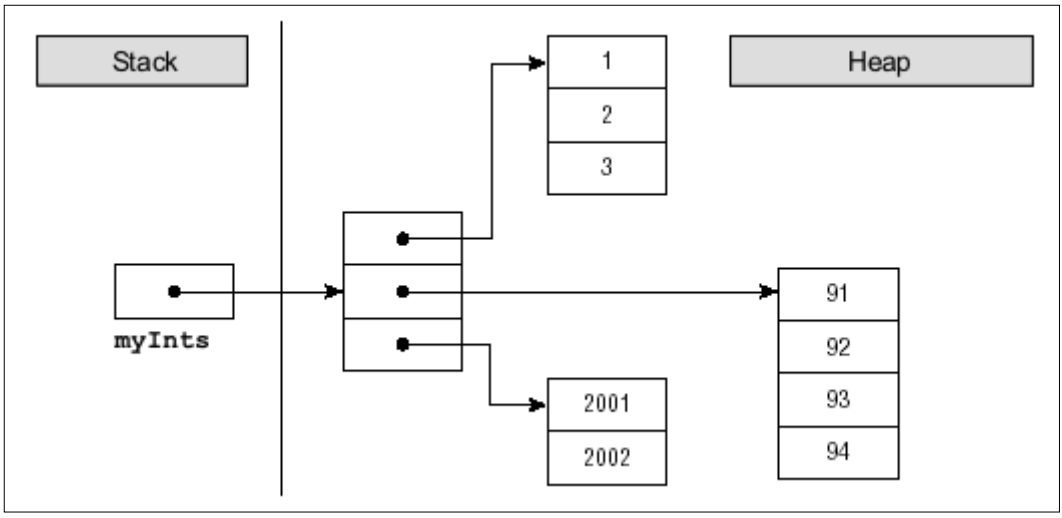
В горния пример инициализираме двумерен масив с цели числа с 2 реда и 4 колони. Във външните фигурни скоби се поставят елементите от първата размерност, т.е. редовете на двумерната матрица. Всеки ред представлява едномерен масив, който се инициализира по познат за нас начин.

Двумерните масиви и паметта

В паметта двумерните и многомерните масиви съхраняват стойностите си в динамичната памет като референция (указател) към област, съдържаща референции към други масиви. На практика всяка променлива от тип масив (едномерен или многомерен) представлява референция към място в динамичната памет, където се съхраняват елементите на масива. Ако масивът е двумерен, неговите елементи са съответно масиви и за тях се пазят референции към динамичната памет, където стоят съответните им елементи. За да си представим визуално това, нека разгледаме следния масив:

```
int[][] myInts = { {1, 2, 3}, {91, 92, 93, 94}, {2001, 2002} };
```

Този масив не е стандартна матрица, защото е с неправоъгълна форма. Той се състои от 3 реда, като всеки от тях има различен брой колони. Това в Java е позволено и след като бъде инициализиран, масивът се представя в паметта по следния начин:



Достъп до елементите на многомерен масив

Матриците имат две размерности и съответно всеки техен елемент се достъпва с помощта на два индекса – един за редовете и един за колоните. Многомерните масиви имат различен индекс за всяка размерност.

Всяка размерност в многомерен започва от индекс нула.

Нека разгледаме следния пример:

```
int[][] matrix = {
  { 1, 2, 3, 4 },
  { 5, 6, 7, 8 },
};
```

Масивът **matrix** има 8 елемента, разположени в 2 реда и 4 колони. Всеки елемент може да се достъпи по следния начин:

```
matrix[0][0]  matrix[0][1]  matrix[0][2]  matrix[0][3]
matrix[1][0]  matrix[1][1]  matrix[1][2]  matrix[1][3]
```

В горния пример виждаме как да достъпим всеки елемент по индекс. Ако означим индекса по редове с **i**, а индекса по колони с **j**, тогава достъпа до елемент от двумерен масив има следния общ вид:

```
matrix[i][j]
```

При многомерните масиви всеки елемент се идентифицира уникално с толкова на брой индекси, колкото е размерността на масива:

```
nDimensionalArray[index1]…[indexN]
```

Дължина на многомерен масив

Всяка размерност на многомерен масив има собствена дължина, която е достъпна по време на изпълнение на програмата. Нека разгледаме следния пример за двумерен масив:

```
int[][] matrix = {
  { 1, 2, 3, 4 },
  { 5, 6, 7, 8 },
};
```

Можем да извлечем броя на редовете на този двумерен масив чрез **matrix.length**. Това на практика е дължината на едномерния масив,

съдържащ референциите към своите елементи (които са също масиви). Извличането на дължината на **i**-ия ред става с **matrix[i].length**.

Отпечатване на матрица – пример

Със следващия пример ще демонстрираме как можем да отпечатваме двумерни масиви на конзолата:

```
// Declare and initialize a matrix of size 2 x 4
int[][] matrix = {
  { 1, 2, 3, 4 }, // row 0 values
  { 5, 6, 7, 8 }, // row 1 values
};

// Print the matrix on the console
for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[0].length; col++) {
      System.out.printf("%d ", matrix[row][col]);
    }
    System.out.println();
}
```

Първо декларираме и инициализираме масива, който искаме да обходим и да отпечатаме на конзолата. Масивът е двумерен и за това използваме един цикъл, който ще се движи по редовете и втори, вложен цикъл, който за всеки ред ще се движи по колоните на масива. За всяка итерация по подходящ начин извеждаме текущия елемент на масива като го достъпваме по неговите два индекса. В крайна сметка, ако изпълним горния програмен фрагмент, ще получим следния резултат:

```
1 2 3 4
5 6 7 8
```

Четене на матрица от конзолата – пример

Нека видим как можем да прочетем двумерен масив (матрица) от конзолата. Това става като първо въведем големините на двете размерности, а след това с два вложени цикъла въвеждаме всеки от елементите му:

```
Scanner input = new Scanner(System.in);
int rows = input.nextInt();
int cols = input.nextInt();

int[][] matrix = new int[rows][cols];

for (int row = 0; row < rows; row++) {
  for (int col = 0; col < cols; col++) {
```

```
      System.out.printf("matrix[%d,%d] = ", row, col);
      matrix[row][col] = input.nextInt();
  }
}

System.out.println(Arrays.deepToString(matrix));
```

Както се вижда от примера, отпечатването на матрица може да стане с метода **Arrays.deepToString()**. Ето как може да изглежда програмата в действие (в случая въвеждаме масив с размер 3 на 2):

```
3
2
matrix[0,0] = 6
matrix[0,1] = 8
matrix[1,0] = 4
matrix[1,1] = 5
matrix[2,0] = 3
matrix[2,1] = 9
[[6, 8], [4, 5], [3, 9]]
```

Максимална площадка в матрица – пример

В следващия пример ще решим една интересна задача: Дадена е правоъгълна матрица с числа. Трябва да намерим в нея максималната подматрица с размер 2 x 2 и да я отпечатаме на конзолата. Под максимална подматрица ще разбираме подматрица, която има максимална сума на елементите, които я съставят. Ето едно примерно решение на задачата:

MaxPlatform2x2.java

```
public class MaxPlatform2x2 {

   public static void main(String[] args) {
      // Declare and initialize the matrix
      int[][] matrix = {
         { 0, 2, 4, 0, 9, 5 },
         { 7, 1, 3, 3, 2, 1 },
         { 1, 3, 9, 8, 5, 6 },
         { 4, 6, 7, 9, 1, 0 }
      };

      // Find the maximal sum platform of size 2 x 2
      int bestSum = Integer.MIN_VALUE;
      int bestRow = 0;
      int bestCol = 0;
      for (int row = 0; row < matrix.length - 1; row++) {
```

```
        for (int col = 0; col < matrix[0].length - 1; col++) {
           int sum = matrix[row][col] + matrix[row][col + 1]
                + matrix[row + 1][col] + matrix[row + 1][col + 1];
           if (sum > bestSum) {
              bestSum = sum;
              bestRow = row;
              bestCol = col;
           }
        }
     }

     // Print the result
     System.out.println("The best platform is:");
     System.out.printf("  %d %d%n",
        matrix[bestRow][bestCol],
        matrix[bestRow][bestCol + 1]);
     System.out.printf("  %d %d%n",
        matrix[bestRow + 1][bestCol],
        matrix[bestRow + 1][bestCol + 1]);
     System.out.printf("The maximal sum is: %d%n", bestSum);
   }
}
```

Ако изпълним програмата, ще се убедим, че работи коректно:

```
The best platform is:
  9 8
  7 9
The maximal sum is: 33
```

Нека сега обясним реализирания алгоритъм. В началото на програмата си създаваме двумерен масив, състоящ се от цели числа. Декларираме помощни променливи **bestSum**, **bestRow**, **bestCol,** които инициализираме с минималните за тях стойности.

В променливата **bestSum** ще пазим текущата максимална сума, а в **bestRow** и **bestCol** текущия ред и колона, които са начало на подматрицата с размери 2 x 2, имаща сума на елементите **bestSum**.

За да достъпим всички елементи на подматрица 2x2 са ни необходими индексите на първия й елемент. Като ги имаме лесно можем да достъпим другите 3 елемента по следния начин:

```
matrix[row][col]
matrix[row][col+1]
matrix[row+1][col]
matrix[row+1][col+1]
```

В горния пример **row** и **col** са индексите на отговарящи на първия елемент на матрица с размер 2 x 2, която е част от матрицата **matrix**.

След като вече разбрахме как да достъпим всички елементи на матрица с размер 2 x 2 можем да разгледаме алгоритъма, по който ще я намерим.

Трябва да обходим всеки елемент от главната матрица до предпоследния ред и предпоследната колона. Забележете, че не обхождаме матрицата от край до край, защото при опит да достъпим **row+1** или **col+1** индекс ще излезем извън границите на масива.

Достъпваме съседните елементи на всеки текущ начален елемент на подматрица с размер 2 x 2 и ги събираме. След това проверяваме дали текущата ни сума е по голяма от текущата най-голяма сума. Ако е така текущата сума става текуща най-голяма сума и текущите индекси стават **bestRow** и **bestCol**. Така след обхождане на главната матрица ще имаме индексите на началния елемент на подматрицата, имаща най-голяма сума.

В края на примера си извеждаме на конзолата по подходящ начин търсената подматрица и нейната сума.

Упражнения

1. Да се напише програма, която създава масив с 20 елемента от целочислен тип и инициализира всеки от елементите със стойност равна на индекса на елемента умножен по 5. Елементите на масива да се изведат на конзолата.
2. Да се напише програма, която чете два масива от конзолата и проверява дали са еднакви.
3. Да се напише програма, която сравнява два масива от тип **char** лексикографски (буква по буква) и проверява кой от двата е по-рано в лексикографската подредба.
4. Напишете програма, която намира максималната редица от еднакви елементи в масив. Пример: {2, 1, 1, 2, 3, 3, **2, 2, 2**, 1} → {2, 2, 2}.
5. Напишете програма, която намира максималната редица от нарастващи елементи в масив. Пример: {3, **2, 3, 4**, 2, 2, 4} → {2, 3, 4}.
6. Да се напише програма, която чете от конзолата две цели числа N и K, и масив от N елемента. Да се намерят тези K елемента, които имат максимална сума.
7. Сортиране на масив означава да подредим елементите му в нарастващ (намаляващ) ред. Напишете програма, която сортира масив. Да се използва алгоритъма "**Selection sort**".
8. Напишете програма, която намира най-често срещания елемент в масив. Пример: {**4**, 1, 1, **4**, 2, 3, **4**, **4**, 1, 2, **4**, 9, 3} → 4 (5 пъти).

9. Да се напише програма, която намира последователност от числа в масив, които имат сума равна на число, въведено от конзолата (ако има такава). Пример: {4, 3, 1, **4, 2, 5**, 8}, S=11 → {4, 2, 5}.

10. Напишете програма, която създава следните квадратни матрици и ги извежда на конзолата във форматиран вид. Размерът на матриците се въвежда от конзолата. Пример за **(4,4)**:

a)

1	**5**	**9**	**13**
2	**6**	**10**	**14**
3	**7**	**11**	**15**
4	**8**	**12**	**16**

b)

1	**8**	**9**	**16**
2	**7**	**10**	**15**
3	**6**	**11**	**14**
4	**5**	**12**	**13**

c)

7	**11**	**14**	**16**
4	**8**	**12**	**15**
2	**5**	**9**	**13**
1	**3**	**6**	**10**

d)*

1	**12**	**11**	**10**
2	**13**	**16**	**9**
3	**14**	**15**	**8**
4	**5**	**6**	**7**

11. Да се напише програма, която създава правоъгълна матрица с размер **(n, m)**. Размерността и елементите на матрицата да се четат от конзолата. Да се намери подматрицата с размер **(3,3)**, която има максимална сума.

12. Да се напише програма, която създава масив с всички букви от латинската азбука. Да се даде възможност на потребител да въвежда дума от конзолата и в резултат да се извеждат индексите на буквите от думата.

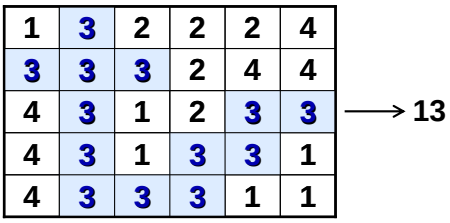
13. Да се реализира двоично търсене **(binary search)** в **сортиран** целочислен масив.

14. Напишете програма, която сортира целочислен масив по алгоритъма **"merge sort"**.

15. Напишете програма, която сортира целочислен масив по алгоритъма **"quick sort"**.

16. Напишете програма, която намира всички прости числа в диапазона [1..10 000 000].

17. Напишете програма, която по подадена матрица намира най-голямата област от еднакви числа. Под област разбираме съвкупност от съседни (по ред и колона) елементи. Ето един пример, в който имаме област, съставена от 13 на брой еднакви елементи със стойност 3:



Решения и упътвания

1. Използвайте масив **int[]** и **for** цикъл.
2. Два масива са еднакви, когато имат еднаква дължина и стойностите на елементите в тях съответно съвпадат. Второто условие можете да проверите с **for** цикъл.
3. При лексикографската наредба символите се сравняват един по един като се започне от най-левия. При несъвпадащи символи по-рано е масивът, чийто текущ символ е по-рано в азбуката. При съвпадение се продължава със следващия символ вдясно. Ако се стигне до края на единия масив, по-краткият е лексикографски по-рано.

   Реализирайте цикъл, който сравнява буквите в масивите на позиции от 0 до дължината на по-късия масив -1, една по една докато намери разлика. Ако бъде намерена разлика, е ясно кой масив е по-рано лексикографски. Ако разлика не бъде намерена, по-късият масив е по-рано лексикографски. Ако масивите съвпадат, никой не от тях е лексикографски по-рано.
4. Сканирайте масива отляво надясно и във всеки един момент пазете в една променлива последните колко позиции (до текущата включително) има еднакви стойности. Пазете най-добрия старт и най-добрата дължина за момента в отделни две променливи.
5. Сканирайте масива отляво надясно и във всеки един момент пазете в променлива колко елемента има в нарастващ ред, които завършват с текущия елемент. Пазете най-добрия старт и най-добрата дължина за момента в отделни две променливи.
6. Сортирайте масива в нарастващ ред и вземете от него последните K елемента.
7. Потърсете в Интернет информация за алгоритъма "**Selection sort**" и негови реализации.
8. Ако сортирате масива, еднаквите му елементи ще бъдат един до друг. Така ако го сканирате отляво надясно и преброите всяка група еднакви елементи от колко елемента се състои, можете да намерите и най-голямата група. Тя съдържа най-често срещания елемент.

9. С два вложени цикъла можете да намерите сумата на всички редици. Единият цикъл ще бележи стартовия елемент, от който започваме да сумираме, а другият – броя сумирани елементи.

10. Помислете за подходящи начини за итерация върху масивите с два вложени цикъла.

    За d) можете да приложите следната стратегия: започвате от позиция (0,0) и се движите надолу N пъти. След това се движите надясно N-1 пъти, след това нагоре N-1 пъти, след това наляво N-2 пъти, след това надолу N-2 пъти и т.н. При всяко преместване слагате в клетката, която напускате поредното число 1, 2, 3, ..., N.

11. Модифицирайте примера за максимална площадка с размер 2 x 2.

12. Задачата можем да решим с масив и два вложени **`for`** цикъла (по буквите на думата и по масива за всяка буква). Задачата има и хитро решение без масив: индексът на дадена буква **`ch`** от азбуката може да се сметне чрез израза: **`(int) ch - (int) 'A'`**.

13. Потърсете в Интернет информация за алгоритъма "**`binary search`**". Какво трябва да е изпълнено, за да използваме този алгоритъм?

14. Потърсете в интернет информация за алгоритъма "**`merge sort`**" и негови реализации.

15. Потърсете в интернет информация за алгоритъма "**`quick sort`**" и негови реализации.

16. Потърсете в интернет информация за "**`sieve of Erathostenes`**".

17. Тази задача е доста по-трудна от останалите. Може да използвате алгоритми за обхождане на граф, известни с названията "DFS" (Depth-first-search) или "BFS" (Breadth-first-search). Потърсете информация и примери за тях в Интернет.

Глава 8. Бройни системи

Автор

Петър Велев

Светлин Наков

В тази тема...

В настоящата тема ще разгледаме начините на работата с различни бройни системи и представянето на числата в тях. Повече внимание ще отделим на представянето на числата в десетична, двоична и шестнадесетична бройна система, тъй като те се използват масово в компютърната техника и в програмирането. Ще обясним и начините за кодиране на числовите данни в компютъра и видовете кодове, а именно: прав код, обратен код, допълнителен код и двоично-десетичен код.

История в няколко реда

Използването на различни бройни системи е започнало още в дълбока древност. Това твърдение се доказва от обстоятелството, че още в Египет са използвани слънчевите часовници, а техните принципи за измерване на времето ползват бройни системи. По-голямата част от историците смятат древноегипетската цивилизация за първата цивилизация, която е разделила деня на по-малки части. Те постигат това, посредством употребата на първите в света слънчеви часовници, които не са нищо друго освен обикновени пръти, забити в земята и ориентирани по дължината и посоката на сянката.

По-късно е изобретен по-съвършен слънчев часовник, който прилича на буквата Т и е градуиран по начин, по който да разделя времето между изгрев и залез слънце на 12 части. Това доказва използването на дванадесетична бройна система в Египет, важността на числото 12 обикновено се свързва и с обстоятелството, че лунните цикли за една година са 12, или с броя на фалангите на пръстите на едната ръка (по три на всеки от четирите пръста, като не се смята палеца).

В днешно време десетичната бройна система е най-разпространената бройна система. Може би това се дължи на улесненията, които тя предоставя на човека, когато той брои с помощта на своите пръсти.

Древните цивилизации са разделили денонощието на по-малки части, като за целта са използвали различни бройни системи, дванадесетични и

шестдесетични съответно с основи – 12 и 60. Гръцки астрономи като Хипарх са използвали астрономични подходи, които преди това са били използвани и от вавилонците в Месопотамия. Вавилонците извършвали астрономичните изчисления в шестдесетична система, която били наследили от шумерите, а те от своя страна да я развили около 2000 г. пр. н. е. Не е известно от какви съображения е избрано точно числото 60 за основа на бройната система, но е важно да се знае че, тази система е много подходяща за представяне на дроби, тъй като числото 60 е най-малкото число, което се дели без остатък съответно на 1, 2, 3, 4, 5, 6, 10, 12, 15, 20 и 30.

Някои приложения на шестдесетичната бройна система

Днес шестдесетичната система все още се използва за измерване на ъгли, географски координати и време. Те все още намират приложение при часовниковия циферблат и сферата на глобуса. Шестдесетичната бройна система е използвана и от Ератостен за разделянето на окръжността на 60 части с цел създаване на една ранна система от географски ширини, съставена от хоризонтални линии, минаващи през известни в миналото места от земята. Един век след Ератостен Хипарх нормирал тези линии, като за целта ги направил успоредни и съобразени с геометрията на Земята. Той въвежда система от линии на географската дължина, в които включват 360 градуса и съответно минават от север до юг и от полюс до полюс. В книгата "Алмагест" (150 г. от н. е.) Клавдий Птолемей доразвива разработките на Хипарх чрез допълнително разделяне на 360-те градуса на географската ширина и дължина на други по-малки части. Той разделил всеки един от градусите на 60 равни части, като всяка една от тези части в последствие била разделена на нови 60 по-малки части, които също били равни. Така получените при деленето части, били наречени partes minutae primae, или "първа минута" и съответно partes minutae secundae, или "втора минута". Тези части се ползват и днес и се наричат съответно "минути" и "секунди".

Кратко обобщение

Направихме кратка историческа разходка през хилядолетията, от която научаваме, че бройните системи са били създадени, използвани и развивани още по времето на шумерите. От изложените факти става ясно защо денонощието съдържа (само) 24 часа, часът съдържа 60 минути, а минутата 60 секунди. Това се дължи на факта, че древните египтяни са разделили по такъв начин денонощието, като са въвели употребата на дванадесетична бройна система. Разделянето на часовете и минутите на 60 равни части, е следствие от работата на древногръцките астрономи, които извършват изчисленията в шестдесетична бройна система, която е създадена от шумерите и използвана от вавилонците.

Бройни системи

До момента разгледахме историята на бройните системи. Нека сега разгледаме какво представляват те и каква е тяхната роля в изчислителната техника.

Какво представляват бройните системи?

Бройните системи са начин за представяне (записване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна **азбука**.

По време на различните етапи от развитието на човечеството, различни бройни системи са придобивали известност. Трябва да се отбележи, че днес най-широко разпространение е получила арабската бройна система. Тя използва цифрите 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, като своя азбука. (Интересен е фактът, че изписването на арабските цифри в днешно време се различава от представените по-горе десет цифри, но въпреки това, те пак се отнасят за същата бройна система т.е. десетичната).

Освен азбука, всяка бройна система има и **основа**. Основата е число, равно на броя различни цифри, използвани от системата за записване на числата в нея. Например арабската бройна система е десетична, защото има 10 цифри. За основа може да се избере произволно число, чиято абсолютна стойност трябва да бъде различна от 0 и 1. Тя може да бъде и реално или комплексно число със знак.

В практическо отношение, възниква въпросът: коя е най-добрата бройна система, която трябва да използваме? За да си отговорим на този въпрос, трябва да решим, как ще се представи по оптимален начин едно число като записване (т.е. брой на цифрите в числото) и брой на цифрите, които използва съответната бройна система т.е. нейната основа. По математически път, може да се докаже, че най-доброто съотношение между дължината на записа и броя на използваните цифри, се постига при основа на бройната система Неперовото число (e = 2,718281828), което е основата на естествените логаритми. Да се работи в система с тази основа, е изключително неудобно, защото това число не може да се представи като отношение на две цели числа. Това ни дава основание да заключим, че оптималната основа на бройната система е 2 или 3. Въпреки, че 3 е по-близо до Неперовото число, то е неподходящо за техническа реализация. Поради тази причина, двоичната бройна система, е единствената подходяща за практическа употреба и тя се използва в съвременните електронноизчислителни машини.

Позиционни бройни системи

Бройните системи се наричат **позиционни**, тогава, когато мястото (позицията) на цифрите има значение за стойността на числото. Това означава, че стойността на цифрата в числото не е строго определена и зависи от това на коя позиция се намира съответната цифра в дадено число. Например в числото 351 цифрата 1 има стойност 1, докато при числото 1024 тя има стойност 1000. Трябва да се отбележи, че основите на бройните системи се прилагат само при позиционните бройни системи. В позиционна бройна система числото $A_{(p)} = (a_{(n)}a_{(n-1)}...a_{(0)},a_{(-1)}a_{(-2)}...a_{(-k)})$ може де се представи във вида:

$$A_{(p)} = \sum_{m=n}^{-k} a_m T_m$$

В тази сума T_m има значение на теглови коефициент за m-тия разряд на числото. В повечето случаи обикновено $T_m = P^m$, което означава, че

$$A_{(p)} = \sum_{m=n}^{-k} a_m P^m$$

Образувано по горната сума, числото $A_{(p)}$ е съставено съответно от цяла си част ($a_{(n)}a_{(n-1)}...a_{(0)}$) и от дробна си част ($a_{(-1)}a_{(-2)}...a_{(-k)}$), където всяко a принадлежи на множеството от цели числа $M=\{0, 1, 2, ..., p-1\}$. Лесно се вижда, че, при позиционните бройни системи стойността на всеки разряд е по-голяма от стойността на предходния разряд (съседния разряд отдясно, който е по-младши) с толкова пъти, колкото е основата на бройната система. Това обстоятелство, налага при събиране да прибавяме единица към левия (по-старшия) разряд, ако трябва да представим цифра в текущия разряд, която е по-голяма от основата. Системите с основи 2, 8, 10 и 16 са получили по-широко разпространение в изчислителната техника, и в следващата таблица е показано съответното представяне на числата от 0 до 15 в тях:

Двоична	Осмична	Десетична	Шестнадесетична
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6

0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Непозиционни бройни системи

Освен позиционни, съществуват и непозиционни бройни системи, при които стойността на всяка цифра е постоянна и не зависи по никакъв начин от нейното място в числото. Като примери за такива бройни системи могат да се посочат съответно римската, гръцката, милетската и др. Като основен недостатък, на непозиционните бройни системи трябва да се посочи това, че чрез тях големите числа се представят неефективно. Заради този си недостатък те са получили по-ограничена употреба. Често това би могло да бъде източник на грешка при определяне на стойността на числата. Съвсем накратко ще разгледаме римската и гръцката бройни системи.

Римска бройна система

Римската бройна система използва следните символи за представяне на числата:

Римска цифра	Десетична равностойност
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Както вече споменахме, в тази бройна система позицията на цифрата не е от значение за стойността на числото, но за нейното определяне се прилагат следните правила:

1. Ако две последователно записани римски цифри, са записани така, че стойността на първата е по-голяма или равна на стойността на втората, то техните стойности се събират. Пример:

   Числото III=3, а числото MMD=2500.

2. Ако две последователно записани римски цифри, са в намаляващ ред на стойностите им, то техните стойности се изваждат. Пример:

   Числото IX=9, числото XML=1040, а числото MXXIV=1024.

Гръцка бройна система

Гръцката бройна система, е десетична система, при която се извършва групиране по петици. Тя използва следните цифри:

Гръцка цифра	Десетична равностойност
I	1
П	5
Δ	10
H	100
X	1 000
M	10 000

Следват примери на числа от тази система:

ГΔ = 50, ГН = 500, ГХ = 5000, ГМ = 50 000.

Двоичната бройна система – основа на електронноизчислителната техника

Двоичната бройна система, е системата, която се използва за представяне и обработка на числата в съвременните електронноизчислителни машини. Главната причина, поради която тя се е наложила толкова широко, се обяснява с обстоятелството, че устройства с две устойчиви състояния се реализират просто, а разходите за производство на двоични аритметични устройства са много ниски.

Двоичните цифри 0 и 1 лесно се представят в изчислителната техника като "има ток" / "няма ток" или като "+5V" и "-5V".

Наред със своите предимства, двоичната система за представяне на числата в компютъра си има и недостатъци. Един от големите практически недостатъци, е, че числата, представени с помощта на тази система са много дълги, т. е. имат голям брой разреди (битове). Това я прави неудобна за

непосредствена употреба от човека. За избягване на това неудобство, в практиката се ползват бройни системи с по-големи основи.

Десетични числа

Числата представени в десетична бройна система, се задават в първичен вид т.е. вид удобен за възприемане от човека. Тази бройна система има за основа числото 10. Числата записани в нея са подредени по степените на числото 10. Младшият разряд (първият отдясно на ляво) на десетичните числа се използва за представяне на единиците ($10^0=1$), следващият за десетиците ($10^1=10$), следващият за стотиците ($10^2=100$) и т.н. Казано с други думи, всеки следващ разряд е десет пъти по-голям от предшестващия го разряд. Сумата от отделните разряди определя стойността на числото. За пример ще вземем числото 95031, което в десетична бройна система се представя като:

$$95031 = (\mathbf{9}x10^4) + (\mathbf{5}x10^3) + (\mathbf{0}x10^2) + (\mathbf{3}x10^1) + (\mathbf{1}x10^0)$$

Представено в този вид, числото 95031 е записано по естествен за човека начин, защото принципите на десетичната система са възприети като фундаментални за хората. Много е важно да се отбележи, че тези подходи важат и за останалите бройни системи. Те имат същата логическа постановка, но тя е приложена за бройна система с друга основа. Последното твърдение, се отнася включително и за двоичната и шестнайсетината бройни системи, които ще разгледаме в детайли след малко.

Двоични числа

Числата представени в тази бройна система, се задават във вторичен вид т.е. вид удобен за възприемане от изчислителната машина. Този вид е малко по-трудно разбираем за човека. За представянето на двоичните числа, се използва двоичната бройна система, която има за основа числото 2. Числата записани в нея са подредени по степените на двойката. За тяхното представяне, се използват само цифрите 0 и 1.

Прието е, когато едно число се записва в бройна система, различна от десетичната, във вид на индекс в долната му част да се отразява, коя бройна система е използвана за представянето му. Например със записа $\mathbf{1110_{(2)}}$ означаваме число в двоична бройна система. Ако не бъде указана изрично, бройната система се приема, че е десетична. Числото се произнася, като се прочетат последователно неговите цифри, започвайки от ляво на дясно (т.е. прочитаме го от старшия към младия разряд "бит").

Както и при десетичните числа, гледано от дясно наляво, всяко двоично число изразява степените на числото 2 в съответната последователност. На младшата позиция в двоично число съответства нулевата степен ($2^0=1$), на втората позиция съответства първа степен ($2^1=2$), на третата позиция съответства втора степен ($2^2=4$) и т.н. Ако числото е 8-битово, степените достигат до седма ($2^7=128$). Ако числото е 16-битово, степените достигат

до петнадесета (2^{15}=32768). Чрез 8 двоични цифри (0 или 1) могат да се представят общо 256 числа, защото 2^8=256. Чрез 16 двоични цифри могат да се представят общо 65536 числа, защото 2^{16}=65536.

Нека даден един пример за числа в двоична бройна система. Да вземем десетичното число **148**. То е съставено от три цифри: **1**, **4** и **8**, и съответства на следното двоично число:

$$\mathbf{10010100_{(2)}}$$

$$\mathbf{148} = (\mathbf{1}x2^7) + (\mathbf{1}x2^4) + (\mathbf{1}x2^2)$$

Пълното представяне на това число е изобразено в следващата таблица:

Число	1	0	0	1	0	1	0	0
Степен	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Стойност	$1x2^7$= 128	$0x2^6$ =0	$0x2^5$ =0	$1x2^4$ =16	$0x2^3$ =0	$1x2^2$ =4	$0x2^1$ =0	$0x2^0$ =0

Последователността от осем на брой нули и единици представлява един **байт**, т.е. това е едно обикновено осем-разредно двоично число. Чрез един байт могат да се запишат всички числа от 0 до 255 включително. Много често това е не достатъчно и затова се използват по няколко последователни байта за представянето на едно число. Два байта образуват т.н. "машинна дума" **(word)**, която отговаря на 16 бита (при 16-разредните изчислителни машини). Освен нея, в изчислителните машини се използва и т.н. "двойна дума" **(double word)** или **(dword),** съответстваща на 32 бита.

Ако едно двоично число завършва на 0, то е четно, а ако завършва на 1, то е нечетно.

Преминаване от двоична в десетична бройна система

При преминаване от двоична в десетична бройна система, се извършва преобразуване на двоичното число в десетично. Всяко число може да се преобразува от една бройна система в друга, като за целта се извършат последователност от действия, които са възможни и в двете бройни системи. Както вече споменахме, числата записани в двоична бройна система се състоят от двоични цифри, които са подредени по степените на двойката. Нека да вземем за пример числото $\mathbf{11001_{(2)}}$. Преобразуването му в десетично се извършва чрез пресмятането на следната сума:

$$\mathbf{11001}_{(2)} = \mathbf{1}x2^4 + \mathbf{1}x2^3 + \mathbf{0}x2^2 + \mathbf{0}x2^1 + \mathbf{1}x2^0 =$$

$$= \mathbf{16}_{(10)} + \mathbf{8}_{(10)} + \mathbf{1}_{(10)} = \mathbf{25}_{(10)}$$

От това следва, че $\mathbf{11001}_{(2)} = \mathbf{25}_{(10)}$

С други думи, всяка една двоична цифра се умножава по 2 на степен, позицията, на която се намира (в двоичното число). Накрая се извършва събиране, на числата, получени за всяка от двоичните цифри, за да се получи десетичната равностойност на двоичното число.

Съществува и още един начин за преобразуване, който е известен като схема на Хорнер. При тази схема се извършва умножение на най-лявата цифра по две и събиране със съседната ѝ вдясно. Този резултат се умножава по две и се прибавя следващата съседна цифра от числото (цифрата вдясно). Това продължава до изчерпване на всички цифри в числото, като последната цифра от числото се добавя без умножаване. Ето един пример:

1001$_{(2)}$ = ((1.2+0).2+0).2+1 = 2.2.2+1 = **9**

Преминаване от десетична към двоична бройна система

При преминаване от десетична в двоична бройна система, се извършва преобразуване на десетичното число в двоично. За целите на преобразуването се извършва делене на две с остатък. Така се получават частно и остатък, който се отделя.

Отново ще вземем за пример числото 148. То се дели целочислено на основата, към която ще преобразуваме (в примера тя е 2). След това, от остатъците получени при деленето (те са само нули и единици), се записва преобразуваното число. Деленето продължава, докато получим частно нула. Ето пример:

148:2=74 имаме остатък 0;

74:2=37 имаме остатък 0;

37:2=18 имаме остатък 1;

18:2=9 имаме остатък 0;

9:2=4 имаме остатък 1;

4:2=2 имаме остатък 0;

2:2=1 имаме остатък 0;

1:2=0 имаме остатък 1;

След като вече семе извършили деленето, записваме стойностите на остатъците в ред, обратен на тяхното получаване, както следва:

10010100

т.е. $148_{(10)}$ = 10010100 $_{(2)}$

Действия с двоични числа

При двоичните числа за един двоичен разряд са в сила аритметичните правила за събиране, изваждане и умножение.

0+0=0	0-0=0	0.0=0
1+0=1	1-0=1	1.0=0
0+1=1	1-1=0	0.1=0
1+1=10	10-1=1	1.1=1

С двоичните числа могат да се извършват и логически действия, като логическо умножение (конюнкция), логическо събиране (дизюнкция) и сума по модул две (изключващо или).

Трябва да се отбележи, че при извършване на аритметични действия над многоразредни числа трябва да се отчита връзката между отделните разреди чрез пренос или заем, когато извършваме съответно събиране или изваждане.

Шестнайсетични числа

При тези числа имаме за основа на бройната система числото 16, което налага да бъдат използвани 16 знака (цифри) за представянето на всички възможни стойности от 0 до 15 включително. Както вече беше показано в една от таблиците в предходните точки, за представянето на шестнайсетичните числа се използват числата от 0 до 9 и латинските букви от A до F. Всяка от тях има съответната стойност:

A=10, B=11, C=12, D=13, E=14, F=15

Като примери за шестнайсетични числа могат да бъдат посочени съответно, D2, 1F2 F1, D1E и др.

Преминаването към десетична система става като се умножи по 16^0 стойността на най-дясната цифра, по 16^1 следващата вляво, по 16^2 следващата вляво и т.н. и накрая се съберат. Например:

$D1E_{(16)} = E*16^0 + 1*16^1 + D*16^2 = 14*1 + 1*16 + 13*256 = 3358_{(10)}$.

Преминаването от десетична към шестнайсетична бройна система става като се дели десетичното число на 16 и се вземат остатъчите в обратен ред. Например:

3358 / 16 = 209 + остатък 14 (E)

209 / 16 = 13 + остатък 1 (1)

13 / 16 = 0 + остатък 13 (D)

Взимаме остатъците в обратен ред и получаваме числото $D1E_{(16)}$.

Бързо преминаване от двоични към шестнайсетични числа

Бързото преобразуване, от двоични в шестнайсетични числа се извършва бързо и лесно, чрез разделяне на двоичното число на групи от по четири бита (разделяне на полубайтове). Ако броят на цифрите в числото не е кратен на четири, то се добавят водещи нули в старшите разреди. След разделянето и евентуалното добавяне на нули, се заместват всички получени групи със съответстващите им цифри. Ето един пример:

Нека да ни е дадено следното число: $1110011110_{(2)}$.

1. Разделяме го на полубайтове и добавяме водещи нули

   Пример: 0011 1001 1110.

2. Заместваме всеки полубайт със съответната шестнайсетична цифра и така получаваме $39E_{(16)}$.

Следователно $1110011110_{(2)} = 39E_{(16)}$.

Представяне на числата

За съхраняване на данните в оперативната памет на електронноизчислителните машини, се използва двоичен код. В зависимост от това какви данни съхраняваме (символи, цели или реални числа с цяла и дробна част) информацията се представя по различен начин. Този начин се определя от типа на данните.

Дори и програмистът на език от високо ниво трябва да знае, какъв вид имат данните разположени в оперативната памет на машината. Това се отнася, и за случаите, когато данните се намират на външен носител, защото при обработката им те се разполагат в оперативната памет.

В тази глава са разгледани начините за представяне и обработка на различни типове данни. Най-общо те се основават на понятията бит, байт и машинна дума.

Бит е една двоична единица от информация, със стойност 0 или 1.

Информацията в паметта се групира в последователности от 8 бита, които образуват един **байт**.

За да бъдат обработени от аритметичното устройство, данните се представят в паметта от определен брой байтове (2, 4 или 8), които образуват машинната дума. Това са концепции, които всеки програмист трябва задължително да знае и разбира.

Представяне на цели числа в паметта

Едно от нещата, на които до сега на обърнахме внимание е знакът на числата. Представянето на цели числата в паметта на компютъра, може да

се извърши по два начина: със знак или без знак. Когато числата се представят със знак се въвежда знаков разред. Той е най-старшият разред и има стойност 1 за отрицателните числа и 0 за положителните. Останалите разреди са информационни и отразяват (съдържат) стойността числото. В случая на числа без знак всички битове се използват за записване на стойността на числото.

Цели числа без знак

За целите числа без знак се заделят по 1, 2, 4 или 8 байта от паметта. В зависимост, от броя на байтовете използвани при представянето на едно число, се образуват обхвати на представяне с различна големина. Посредством *n* на брой бита могат да се представят цели числа без знак в обхвата $[0, 2^n-1]$. Следващата таблица, показва обхвата от стойности на целите числа без знак:

Брой байтове за представяне на числото в паметта	Обхват	
	Запис чрез порядък	Обикновен запис
1	$0 \div 2^8-1$	0 ÷ 255
2	$0 \div 2^{16}-1$	0 ÷ 65 535
4	$0 \div 2^{32}-1$	0 ÷ 4 294 967 295
8	$0 \div 2^{64}-1$	0 ÷ 9 223 372 036 854 775 807

Ще покажем пример при еднобайтово и двубайтово представяне на числото 158, което се записва в двоичен вид като $10011110_{(2)}$:

1. Представяне с 1 байт:

1	0	0	1	1	1	1	0

2. Представяне с 2 байта:

0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0

Представяне на отрицателни числа

За отрицателните числа се заделят по един, два или четири байта от паметта на компютъра, като най-старшият разред има значение на знаков и ни носи информация за знака на числото. Както вече споменахме, когато знаковият бит има стойност 1 числото е отрицателно, а в противен случай – положително. Следващата таблица, показва обхвата от стойности на целите числа със знак:

Брой байтове за представяне на числото в паметта	Обхват	
	Запис чрез порядък	Обикновен запис
1	$-2^7 \div 2^7-1$	-128 ÷ 127
2	$-2^{15} \div 2^{15}-1$	-32 768 ÷ 32 767
4	$-2^{31} \div 2^{31}-1$	-2 147 483 648 ÷ 2 147 483 647
8	$-2^{63} \div 2^{63}-1$	-9 223 372 036 854 775 808 ÷ 9 223 372 036 854 775 807

За кодирането на отрицателните числа, се използват прав, обратен и допълнителен код. И при трите представяния целите числа със знак са в границите: $[-2^{n-1}, 2^{n-1}-1]$. Положителните числа винаги се представят по един и същи начин и за тях правият, обратният и допълнителният код съвпадат.

Прав код: Правият код е най-простото представяне на числото. Старшият бит е знаков, а в оставащите битове е записана абсолютната стойност на числото. Ето няколко примера:

Числото 3 в прав код се представя в осембитово число като 00000011.

Числото -3 в прав код се представя в осембитово число като 10000011.

Обратен код: Получава се от правия код на числото, чрез инвертиране (заместване на всички нули с единици и единици с нули). Този код не е никак удобен за извършването на аритметичните действия събиране и изваждане, защото се изпълнява по различен начин, когато се налага изваждане на числа. Освен това се налага знаковите битове да се обработват отделно от информационните. Този недостатък се избягва с употребата на допълнителен код, при който вместо изваждане се извършва събиране с отрицателно число. Последното е представено чрез неговото допълнение т.е. разликата между 2^n и самото число. Пример:

Числото -127 в прав код се представя като 1 1111111, а в обратен код като 1 0000000.

Числото 3 в прав код се представя като 0 0000011, а в обратен код има вида 0 1111100.

Допълнителен код: Допълнителният код е число в обратен код, към което е прибавена (чрез събиране) единица. Пример:

Числото -127 представено в допълнителен код има вида 1 0000001.

Двоично-десетичен код: Известен е още като BCD код (Binary Coded Decimal). При този код в един байт се записват по две десетични цифри. Това се постига, като чрез всеки полубайт се кодира една десетична цифра. Числа представени чрез този код могат да се пакетират т.е. да се

представят в пакетиран формат. Ако представим една десетична цифра в един байт се получава непакетиран формат.

Съвременните микропроцесори използват един или няколко от разгледаните кодове за представяне на отрицателните числа, като най-разпространеният начин е представянето в допълнителен код.

Типовете int и long в Java

Както знаем, в Java има четири целочислени типа данни със знак и те са **byte**, **short**, **int**, и **long**. За представянето на променливи от тези типове се използват двоични числа в допълнителен код. В зависимост от броя байтове, които се заделят в паметта за тези типове, се определя и съответният диапазон от стойности, които те могат да заемат.

Съществено ограничение на езика Java е, че не поддържа цели числа без знак. Най-широко използваният целочислен тип е **int**. Той се представя като 32-битово число в допълнителен код и приема стойности в интервала $[-2^{31}, 2^{31}-1]$. Променливите от този тип най-често се използват за управление на цикли, индексиране на масиви и други целочислени изчисления. В следващата таблица е даден пример за декларация на променливи от тип **int**:

```
int decimalValue = 25;
int octoInt = 0235;
```

Типът **long** е най-големият целочислен тип със знак в Java. Той има размерност 64 бита (8 байта). При присвояване на стойности на променливите от тип **long** се използват латинските букви "**l**" или "**L**", които се поставят в края на целочисления литерал. Поставен на това място, този модификатор означава, че литералът има стойност от тип **long**. Това се прави, защото по подразбиране целочислените литерали са от тип **int**. В следващия пример декларираме и присвояваме 64-битови цели числа на променливи от тип **long**:

```
long longValue = 9223372036854775807L;
long newLongValue = 9321456990543236891;
```

Важно условие е да се внимава да не бъде надхвърлен обхватът на представимите числа и за двата типа.

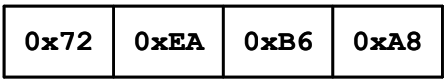
Представянията Big-Endian и Little-Endian

При цели числа, които се записват в повече от един байт, има два варианта за наредба на байтовете в паметта:

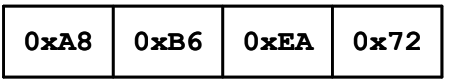
- Little-Endian (LE) – байтовете се подреждат от ляво надясно от най-младшия към най-старшия. Това представяне се използва при Intel x86, Intel x64 микропроцесорните архитектури.

- Big-Endian (BE) – байтовете се подреждат от ляво надясно от най-старшия към най-младшия. Това представяне се използва при PowerPC, SPARC и ARM микропроцесорните архитектури.

Ето един пример: числото $A8B6EA72_{(16)}$ се представя в двете наредби на байтовете по следния начин:



Little-Endian (LE)
for 0xA8B6EA72



Big-Endian (BE)
for 0xA8B6EA72

Java използва Big-Endian представянето, което е типично за хардуера на Sun Microsystems, и това трябва да се съобразява при обмяната на числови данни с други системи, които не са Java базирани.

Представяне на реални числа с плаваща запетая

Реалните числа са съставени от цяла и дробна част. В компютрите, те се представят като числа с плаваща запетая. Всъщност това представяне идва от възприетия от водещите производители на микропроцесори Standard for Floating-Point Arithmetic (IEEE 754). Повечето хардуерни платформи и езици за програмиране позволят или изискват изчисленията да се извършват съгласно изискванията на този стандарт. Стандартът определя:

- Аритметични формати: набор от двоични и десетични данни с плаваща запетая, които са съставени от краен брой цифри
- Формати за обмен: кодировки (битови низове), които могат да бъдат използвани за обмен на данни в една ефективна и компактна форма
- Алгоритми за закръгляване: методи, които се използват за закръгляване на числата по време на изчисления
- Операции: аритметика и други операции на аритметичните формати
- Изключения: представляват сигнали за извънредни случаи като деление на нула, препълване и др.

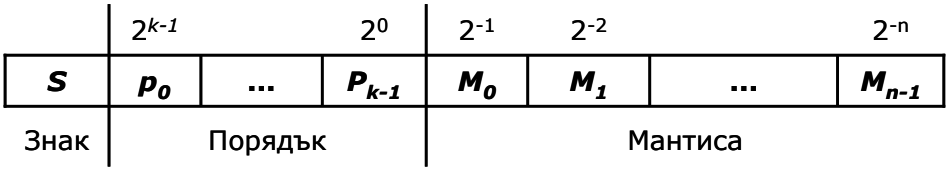
Съгласно IEEE-754 стандарта произволно реално число R може да бъде представено във вида:

$$R = M * q^p$$

където M е **мантисата** на числото, а p е **порядъкът** му (**експонента**), и съответно q е основа на бройната система, в която е представено числото. Мантисата трябва да бъде положителна или отрицателна правилна дроб т.е. $|M|<1$, а порядъкът – положително или отрицателно цяло число.

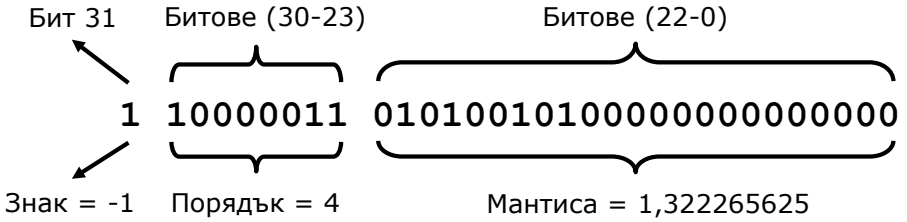
При посочения начин на представяне на числата, всяко число във формат с плаваща запетая, ще има следния обобщен вид $\pm 0,M*q^{\pm p}$.

В частност, когато представяме числата във формат с плаваща запетая чрез двоична бройна система, ще имаме $R = M * 2^p$. При това представяне на реалните числа в паметта на компютъра, след промяна на порядъка се стига и до изместване "плаване" на десетичната запетая в мантисата. Форматът на представянето с плаваща запетая, има полулогаритмична форма. Той е изобразен нагледно на следващата фигура:



Представяне на числа с плаваща запетая – пример

Нека дадем един пример за представяне на число с плаваща запетая в паметта. Искаме да запишем числото -21,15625 в 32-битов (single precision) floating-point формат по стандарта IEEE-754. При този формат се използват 23 бита за мантиса, 8 бита за експонента и 1 бит за знак на числото. Представянето на числото е следното:



Знакът на числото е отрицателен, т. е. мантисата има отрицателен знак:

$S = -1$

Порядъкът (експонентата) има стойност 4 (записана в допълнителен код):

$p = (2^0 + 2^1 + 2^7) - 127 = (1+2+128) - 127 = 4$

За преминаване към истинската стойност изваждаме 127 от стойността на допълнителния код, защото работим с 8 бита ($127 = 2^8-1$).

Мантисата има следната стойност (без да взимаме предвид знака):

$$M = 1 + 2^{-2} + 2^{-4} + 2^{-7} + 2^{-9} =$$
$$= 1 + 0,25 + 0,0625 + 0,0078125 + 0,001953125 =$$
$$= 1,322265625$$

Забелязахте ли, че добавихме единица, която липсва в двоичния запис на мантисата? Това е така, защото мантисата винаги е нормализирана и започва с единица, която се подразбира.

Стойността на числото се изчислява по формулата $R = M * 2^p$, която в нашия пример добива вида:

$$R = -1{,}3222656 * 2^4 = -1{,}322265625 * 16 = -21{,}1562496 \approx -21{,}15625$$

Нормализация на мантисата

За по-пълното използване на разрядната решетка мантисата трябва да съдържа единица в старшия си разред. Всяка мантиса удовлетворяваща това условие са нарича **нормализирана**. При IEEE-754 стандарта единицата в цялата част на мантисата се подразбира, т.е. мантисата е винаги число между 1 и 2.

Ако по време на изчисленията се достигне до резултат, който не удовлетворява това условие, то имаме нарушение на нормализацията. Това изисква, преди да се пристъпи към по-нататъшна обработка на числото то да бъде нормализирано, като за целта се измества запетаята в мантисата и след това се извършва съответна промяна на порядъка.

Типовете float и double в Java

В Java разполагаме с два типа данни за представяне на числата с плаваща запетая. Типът **`float`** е 32-битово реално число с плаваща запетая, за което е прието да се казва, че има единична точност (single precision floating-point number). Типът **`double`** е 64-битово реално число с плаваща запетая, за което е прието да се казва, че има двойна точност (double precision floating-point number). Тези реални типове данни и аритметичните операции върху тях съответстват на спецификацията, определена от стандарта IEEE 754-1985.

При тип **`float`** имаме мантиса, която съхранява 8-9 значещи цифри, докато при тип **`double`** тя съхранява 16-17 значещи цифри. Останалите битове се използват за задаването на знаците на мантисата и стойността на порядъка. Типът **`double`** освен с по-голям брой значещи цифри разполага и с по-голям порядък т.е. обхват на приеманите стойности. Ето един пример за декларация на променливи от тип **`float`** и тип **`double`**:

```
float total = 5.0f;
float result = 5.0F;
double sum = 10.0;
double div = 35.4/3.0;
```

Важно е да се знае, че в Java по подразбиране числата с плаваща запетая са от тип **`double`**.

Много от математическите операции могат да дадат резултати, които нямат конкретна числена стойност като например стойността "+/- безкрайност" или стойността NaN (което означава "Not a Number"), които не представляват числа. Ето един пример:

```
double d = 0;
System.out.println(d);
```

```
System.out.println(1/d);
System.out.println(-1/d);
System.out.println(d/d);
```

Ако го изпълним, ще получим следния резултат:

```
0.0
Infinity
-Infinity
NaN
```

Ако изпълним горния код с тип int вместо double, ще получим **java.lang.ArithmeticException**, защото целочисленото деление на 0 е непозволена операция.

Стриктен режим на изчисленията с плаваща запетая

Изчисленията с плаваща запетая могат да се изпълняват и в стриктен режим. Стриктната аритметика при числата с плаваща запетая следва строги правила за операциите, които гарантират, че ще получим един и същ резултат от изчисленията при изпълнение на програмата на различни версии на виртуалната машина.

Ако се налага да гарантирате побитова еднаквост на резултата във всяка реализация на виртуалната машина, трябва да ползвате модификатора **strictfp**. Той може да се приложи към клас, интерфейс или метод. Повече информация относно този въпрос, може да се намери в "The Java Language Specification": http://java.sun.com/docs/books/jls/.

Точност на числата с плаваща запетая

Точността на резултатите от изчисленията при работа с числа с плаваща запетая зависят от следните параметри:

1. Точността на представяне на числата.
2. Точността на използваните числени методи.
3. Стойностите на грешките, получени при закръгляване и др.

При числата, представени посредством плаваща запетая, стойностите на абсолютната и относителната грешка се определят както следва:

Абсолютна грешка:

$$\Delta R = 2^p * \Delta M = 2^p * 2^{-n-1}$$

Относителна грешка:

$$\varepsilon = \frac{\Delta R}{R} = \frac{2^p * 2^{-n-1}}{2^p * M} = \frac{2^{-n-1}}{M}$$

Поради това, че числата се представят с някаква точност, при изчисление и резултатите имат съответната точност. Нека като пример да разгледаме следния програмен фрагмент:

```
double sum = 0.0;
for(int i=1; i<=10; i++) {
   sum += 0.1;
}
System.out.println(sum);
```

По време на неговото изпълнение, в цикъл добавяме стойността 1/10 в променливата `sum`. Очаква се, че при изпълнението на тази програма ще получим резултат 1.0, но в действителност програмата ще изведе стойност, много близка до вярната, но все пак различна:

```
0.9999999999999999
```

Причината за това е, че числото 0.1 няма точно представяне в типа **`double`** и се представя със закръгляне. Нека заменим **`double`** с **`float`**:

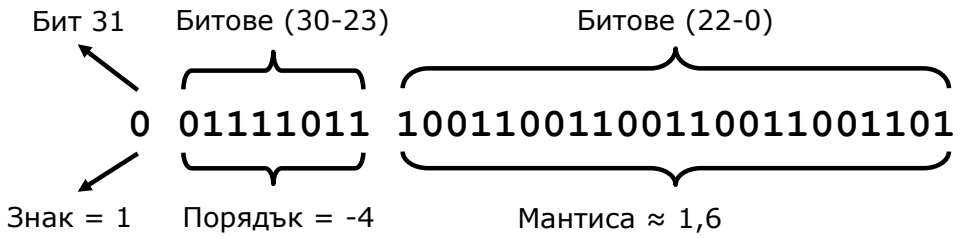
```
float sum = 0.0f;
for(int i=1; i<=10; i++) {
   sum += 0.1f;
}
System.out.println(sum);
```

Ако изпълним горния код ще получим съвсем друга сума:

```
1.0000001
```

Причината за това отново е закръглянето, което е извършено този път нагоре, а не надолу, както в предходния пример.

Ако направим разследване, ще се убедим, че числото 0.1 се представя в типа **`float`** по следния начин:



Всичко изглежда коректно с изключение на мантисата, която има стойност, малко по-голяма от 1.6, а не точно 1.6, защото това число не може да се представи като сума от степени на 2. Грешката настъпва не при събирането, а още преди това – при записването на 0.1 в типа **`float`**.

Числа с фиксирана запетая

В някои езици за програмиране съществуват и числа с **фиксирана запетая**, например типът **decimal** в C#, типът **money** в SQL Server и типът **number(10,2)** в Oracle. В Java за съжаление няма такъв примитивен тип.

Ако се нуждаем от точни изчисления (например за счетоводни или финансови цели), можем да ползваме класа **BigDecimal**, който не губи точност при пресмятанията, но за сметка на това работи чувствително по-бавно от **float** и **double**. Ето как изглежда в нов вариант нашата програма, която правеше грешки при сумиране на числа:

```
import java.math.BigDecimal;

public class Precision {
   public static void main(String[] args) {
      double sum = 0.0d;
      BigDecimal bdValue = new BigDecimal("0.1");
      BigDecimal bdSum = new BigDecimal("0.0");
      for(int i=1; i<=10; i++) {
         sum += 0.1d;
         bdSum = bdSum.add(bdValue);
      }
      System.out.println("Double sum is: " + sum);
      System.out.println("BigDecimal sum is: " + bdSum);
   }
}
```

По време на нейното изпълнение, отново добавяме в цикъл стойността една десета в променливата `sum` от тип `double` и обектната променливата от тип **BigDecimal**. След изпълнението на програмата ще получим:

```
Double sum is: 0.9999999999999999
BigDecimal sum is: 1.0
```

Примерът ни доказва, че **BigDecimal** не прави грешка при събирането на 0.1 десет пъти, за разлика от тип **double**.

Упражнения

1. Превърнете числата 151, 35, 43, 251 и -0,41 в двоична бройна система.
2. Превърнете числото **1111010110011110(2)** в шестнадесетична и десетична бройна система.
3. Превърнете шестнайсетичните числа 2A3E, FA, FFFF, 5A0E9 в двоична и десетична бройна система.
4. Да се напише програма, която преобразува десетично число в двоично.

5. Да се напише програма, която преобразува двоично число в десетично.
6. Да се напише програма, която преобразува десетично число в шестнадесетично.
7. Да се напише програма, която преобразува шестнадесетично число в десетично.
8. Да се напише програма, която преобразува шестнадесетично число в двоично.
9. Да се напише програма, която преобразува двоично число в шестнадесетично.
10. Да се напише програма, която преобразува двоично число в десетично по схемата на Хорнер.
11. Да се напише програма, която преобразува римските числа в арабски.
12. Да се напише програма, която преобразува арабските числа в римски.
13. Да се напише програма, която определя (отпечатва) стойността на мантисата, знака на мантисата и стойността на експонентата за числа от тип **`float`** и **`double`**.

Решения и упътвания

1. Използвайте методите за превръщане от една бройна система в друга. Можете да сверите резултатите си с калкулатора на Windows, който поддържа работа с бройни системи след превключване в режим "Scientific".
2. Погледнете упътването за предходната задача.
3. Погледнете упътването за предходната задача.
4. Правилото е "делим на 2 и долепяме остатъците в обратен ред". За делене с остатък използваме оператора **`%`**.
5. Започнете от сума 0. Умножете най-десния бит с 1 и го прибавете към сумата. Следващия бит вляво умножете по 2 и добавете към сумата. Следващия бит отляво умножете по 4 и добавете към сумата и т.н.
6. Правилото е "делим на основата на системата (16) и долепяме остатъците в обратен ред". Трябва да си напишем метод за отпечатване на шестнайсетична цифра по дадена стойност между 0 и 15.
7. Започнете от сума 0. Умножете най-дясната цифра с 1 и я прибавете към сумата. Следващата цифра вляво умножете по 16 и я добавете към сумата. Следващата цифра вляво умножете по 16*16 и я добавете към сумата и т.н.

8. Ползвайте бързия начин за преминаване между шестнайсетична и двоична бройна система (всяка шестнайсетична цифра съответства на 4 двоични бита).
9. Ползвайте бързия начин за преминаване между двоична и шестнайсетична бройна система (всяка шестнайсетична цифра съответства на 4 двоични бита).
10. Приложете директно схемата на Хорнер.
11. Сканирайте цифрите на римското число отляво надясно и ги добавяйте към сума, която първоначално е инициализирана с 0. При обработката на всяка римска цифра я взимайте с положителен или отрицателен знак в зависимост от следващата цифра (дали има по-малка или по-голяма десетична стойност).
12. Разгледайте съответствията на числата от 1 до 9 с тяхното римско представяне с цифрите "I", "V" и "X":

    1 -> I
    2 -> II
    3 -> III
    4 -> IV
    5 -> V
    6 -> VI
    7 -> VII
    8 -> VIII
    9 -> IX

    Имаме абсолютно аналогични съответствия на числата 10, 20, ..., 90 с тяхното представяне с римските цифри "X", "L" и "C", нали? Имаме аналогични съответствия между числата 100, 200, ..., 900 и тяхното представяне с римските цифри "C", "D" и "M" и т.н.

    Сега сме готови да преобразуваме числото N в римска бройна система. То трябва да е в интервала [1...3999], иначе съобщаваме за грешка. Първо отделяме хилядите (N / 1000) и ги заместваме с римския им еквивалент. След това отделяме стотиците ((N / 100) % 10) и ги заместваме с римския им еквивалент и т.н.
13. Използвайте специалните методи за извличане на побитовото представяне на дадено реално число **`Float.floatToRawIntBits()`** и **`Double.doubleToRawLongBits()`**, след което използвайте подходящи побитови операции (измествания и битови маски).

Глава 9. Методи

Автор

Николай Василев

В тази тема...

В настоящата тема ще се запознаем подробно с това какво е метод и защо трябва да използваме методи. Ще разберем как се декларират методи и какво е сигнатура на метод. След като приключим темата, ще знаем как да създадем собствен метод и съответно как да го използваме (извикваме) в последствие. Ще разберем как можем да използваме параметри в методи и как да върнем резултат от метод. Накрая ще препоръчаме някои утвърдени практики при работата с методи.

Всичко това ще бъде подкрепено с подробно обяснени примери и допълнителни задачи, с които читателят ще може да упражни наученото в тази глава.

Подпрограмите в програмирането

В ежедневието ни, при решаването на даден проблем, особено, ако е по-сложен, прилагаме принципа на древните римляни - "Разделяй и владей". Съгласно този принцип, проблемът, който трябва да решим, се разделя на множество подпроблеми. Самостоятелно разгледани, те са по-ясно дефинирани и по-лесно решими, в сравнение с търсенето на решение на изходния проблем като едно цяло. Накрая, от решенията на всички подпроблеми, създаваме решението на цялостния проблем.

По същата аналогия, когато пишем дадена програма, целта ни е с нея да решим конкретна задача. За да го направим ефективно и да улесним работата си, прилагаме принципа "Разделяй и владей". Разбиваме поставената ни задача на подзадачи, разработваме решения на тези подзадачи и накрая ги "сглобяваме" в една програма. Решенията на тези подзадачи наричаме подпрограми (subroutines).

В някои езици за програмиране подпрограмите могат да се срещнат под наименованията функции (functions) или процедури (procedures). В Java, те се наричат методи (methods).

Какво е "метод"?

Метод (method) е съставна част от програмата, която решава даден проблем.

В методите се извършва цялата обработка на данни, която програмата трябва да направи, за да реши поставената задача. Те са мястото, където се извършва реалната работа. Затова можем да ги приемем като строителен блок на програмата. Съответно, имайки множество от простички блокчета – отделни методи, можем да създаваме големи програми, с които да решим сложни проблеми. Ето как изглежда един метод за намиране лице на правоъгълник например:

```
public static double getRectagnleArea(double width, double height) {
   double area = width * height;
   return area;
}
```

Защо да използваме методи?

Има много причини, които ни карат да използваме методи. Ще разгледаме някои от тях и с времето ще се убедите, че методите са нещо, без което не можем, ако искаме да програмираме сериозно.

По-добро структуриране и по-добра четимост

При създаването на една програма, е добра практика да използваме методи, за да я направим по-добре структурирана и по-лесно четима не само за нас, но и за други хора.

Довод за това е, че за времето, през което съществува една програма, само 20% от усилията, които се заделят за нея, се състоят в създаване и тестване на кода. Останалата част е за поддръжка и добавяне на нови функционалности към началната версия. В повечето случаи, след като веднъж кодът е написан, той не се поддържа и модифицира само от създателя му, но и от други програмисти. Затова е добре той да е добре структуриран и лесно четим.

Избягване на повторението на код

Друга причина, заради която е добре да използваме методи е, че по този начин избягваме повторението на код. Това е пряко свързано със следващата точка – преизползване на кода.

Преизползване на кода

Добър стил на програмиране е, когато използваме даден код повече от един или два пъти в програмата ни, да го дефинираме като отделен метод, за да

можем да го изпълняваме многократно. По този начин освен, че избягваме повторението на код, програмата ни става по-четима и по-добре структурирана.

Деклариране, имплементация и извикване на собствен метод

Преди да продължим по-нататък, ще направим разграничение между три действия свързани със съществуването на един метод – деклариране, имплементация (създаване) и извикване на метод.

Деклариране на метод наричаме регистрирането на метода, за да бъде разпознаван в останалата част на Java-света.

Имплементация (създаване) на метода, е реалното написване на кода, който решава конкретната задача, заради която се създава метода. Този код се съдържа в самия метод.

Извикване е процесът на стартиране на изпълнението, на вече декларирания и създаден метод, от друго място на програмата, където трябва да се реши проблемът, който нашият метод решава.

Деклариране на собствен метод

Преди да се запознаем как можем да декларираме метод, трябва да знаем къде е позволено да го направим.

Къде е позволено да декларираме метод

Въпреки, че формално все още не сме запознати как се декларира клас, от примерите, които сме разглеждали до сега в предходните глави, знаем, че всеки клас има отваряща и затваряща фигурни скоби – "**{**" и "**}**", между които пишем програмния код. Повече подробности за това, ще научим в главата "Дефиниране на класове", но го споменаваме тук, тъй като един метод може да съществува само ако е деклариран **между** отварящата и затварящата скоби на даден клас – "**{**" и "**}**". Също така методът, трябва да бъде деклариран **извън** имплементацията на друг метод (за това малко по-късно).

Можем да декларираме метод единствено в рамките на даден клас – между отварящата "{" и затварящата "}" му скоби.

Най-очевидния пример за това е методът **main()** – винаги го декларираме между отварящата и затварящата скоба на нашия клас, нали?

HelloJava.java

```
public class HelloJava { // Opening brace of the class

   // Declaring our method between the class braces
   public static void main(String[] args) {
      System.out.println("Hello Java!");
   }

} // Closing brace of the class
```

Декларация на метод

Декларирането на метода, представлява **регистриране** на метода в нашата програма. То става по следния начин:

```
[public] [static] <return_type> <method_name>([<param_list>])
```

Задължителните елементи в декларацията на един метод са:

- Тип на връщаната от метода стойност – **<return_type>**.
- Име на метода – **<method_name>**.
- Списък с параметри на метода – **<param_list>** – съществува само ако метода има нужда от тях в процеса на работата си.

За онагледяване на това, можем да погледнем **main()** метода в примера **HelloJava** от предходната секция:

```
public static void main(String[] args)
```

При него, типа на връщаната стойност е **void** (т.е. метода не връща резултат), името му е **main**, следвано от кръгли скоби, в които има списък с параметри, състоящ се от един параметър – масивът **String[] args**.

Последователността, в която трябва да се поставят отделните елементи от декларацията на метода е строго определена. Винаги на първо място е типът на връщаната стойност **<return_type>**, следвана от името на метода **<method_name>** и накрая, списък с параметри **<param_list>** ограден с кръгли скоби – "**(**" и "**)**".

При деклариране на метод, спазвайте последователността, в която се описват основните му характеристики: първо тип на връщана стойност, след това име на метода и накрая списък от параметри ограден с кръгли скоби.

Списъкът от параметри може да е празен (тогава просто пишем "`()`" след името на метода). Дори методът да няма параметри, кръглите скоби трябва да присъстват в декларацията му.

Кръглите скоби – "(" и ")", винаги следват името на метода, независимо дали той е с или без параметри.

За момента, ще пропуснем разглеждането какво е `<return_type>` и само ще кажем, че на това място трябва да стои ключовата дума `void`, която указва, че методът не връща никаква стойност. По-късно в тази глава, ще видим какво представлява и какво можем да поставим на нейно място.

Думите `public` и `static` в описанието на декларацията по-горе са незадължителни и имат специално предназначение, което ще разгледаме по-късно в тази глава. До края на тази глава ще разглеждаме методи, които винаги имат `static` в декларацията си. Повече за методи, които не са декларирани като `static`, ще говорим в главата "Дефиниране на класове".

Сигнатура на метод

Преди да продължим с основните елементи от декларацията на метода, трябва да кажем нещо много важно. В обектно-ориентираното програмиране, начинът, по който еднозначно се разпознава един метод е чрез двойката елементи от декларацията му – име на метода и списък от неговите параметри. Тези два елемента определят така наречената **спецификация на метода** (някъде в литературата се среща и като **сигнатура на метода**).

Java като език за обектно-ориентирано програмиране, също разпознава еднозначно различните методи, използвайки тяхната спецификация – името на метода `<method_name>` и списъкът с параметрите на метода – `<param_list>`.

Трябва да обърнем внимание, че типът на връщаната стойност на един метод е част от декларацията му, но не е част от сигнатурата му.

Това, което идентифицира един метод, е неговата сигнатура. Връщаният тип не е част от нея. Причината е, че ако два метода се различават само по връщания тип, то не може еднозначно да се идентифицира кой метод трябва да бъде извикан.

По-подробен пример, защо типа на връщаната стойност не е част от сигнатурата на метода ще разгледаме по-късно в тази глава.

Име на метода

Всеки метод, решава някаква подзадачка от цялостния проблем, с който се занимава програмата ни. Когато създаваме програмата и стигнем до

подпроблема, който този метод решава, ние извикаме (стартираме) метода, използвайки името му.

В примера показан по-долу, името на метода е **printLogo**:

```
public static void printLogo() {
   System.out.println("Sun Microsystems");
   System.out.println("www.sun.com");
}
```

Правила за създаване на име на метод

Добре е, когато декларираме името на метода, да спазваме правилата за именуване на методи, препоръчани ни от Sun:

- Името на метода трябва да започва с малка буква.
- Трябва да се прилага правилото **camelCase**, т.е. всяка нова дума, която се долепя в задната част на името на метода, започва с главна буква.
- Имената на методите е добре да бъдат съставени от глагол или от глагол и съществително име.

Нека отбележим, че тези правила не са задължителни, а препоръчителни. Но принципно, ако искаме форматирането на кода ни да е като на всички Java-програмисти по света е добре да спазваме конвенциите на Sun.

Ето няколко примера:

```
print
getName
playMusic
setUserName
```

Освен това, името на метода трябва да описва неговата цел. Идеята е, ако човек, който не е запознат с програмата ни, прочете името на метода, да добие представа какво прави този метод, без да се налага да разглежда кода му.

При определяне на името на метод се препоръчва да се спазват следните правила:

- **Името на метода трябва да описва неговата цел.**
- **Името на метода трябва да започва с малка буква.**
- **Трябва да се прилага правилото camelCase.**
- **Името на метода трябва да е съставено от глагол или от двойка - глагол и съществително име.**

Модификатори (modifiers)

Модификатор (modifier) наричаме ключова дума в езика Java, която дава допълнителна информация на компилатора за даден код.

Модификаторите, с които срещнахме до момента са **public** и **static**. Тук ще опишем на кратко какво представляват те. Детайлно обяснение за тях, ще бъде дадено по-късно в главата "Дефиниране на класове".

```
public static void printLogo() {
   System.out.println("Sun Microsystems");
   System.out.println("www.sun.com");
}
```

public е специален вид модификатор, наречен **модификатор за достъп (access modifier)**. Той се използва, за да укаже, че извикването на метода може да става от кой да е Java-клас, независимо къде се намира той.

Друг пример за модификатор за достъп, който може да срещнем е модификатора **private**. Като предназначение, той е противоположен на **public**, т.е. ако един метод бъде деклариран с модификатор за достъп **private**, то този метод не може да бъде извикан извън класа, в който е деклариран.

За момента, единственото, което трябва да научим е, че в декларацията си един метод може да има не повече от един модификатор за достъп.

Когато един метод притежава ключовата дума **static**, в декларацията си, наричаме метода **статичен**. Това означава, че този метод може да бъде извикан от кой да е друг метод, независимо дали другият метод е статичен или не.

Имплементация (създаване) на собствен метод

След като декларираме метода, следва да напишем неговата имплементация. Както обяснихме по-горе, **имплементацията (създаването)** се състои в процеса на написването на кода, който ще бъде изпълнен при извикването на метода. Този код трябва да бъде поставен в тялото на метода.

Тяло на метод

Тяло на метод наричаме програмния код, който се намира между фигурните скоби "**{**" и "**}**", следващи непосредствено декларацията на метода.

```
public static <return_type> <method_name>(<parameters_list>) {
   // ... code goes here – in the method's body ...
}
```

Реалната работа, която методът извършва, се намира именно в тялото на метода. В него трябва да бъде описан алгоритъмът, по който методът решава поставения проблем.

Пример, за тяло на метод сме виждали много пъти, но сега ще изложим отново един:

```
public static void printLogo() { // Method's body start here
   System.out.println("Sun Microsystems");
   System.out.println("www.sun.com");
} // ... And finishes here
```

Преди да приключим със секцията за тяло на метод, трябва отново да обърнем внимание на едно от правилата, къде може да се декларира метод:

Метод НЕ може да бъде деклариран в тялото на друг метод.

Локални променливи

Когато декларираме променлива в тялото на един метод, я наричаме **локална променлива (local variable)** за метода. Когато именуваме една променлива трябва да спазваме правилата за идентификатори в Java (вж. глава "Примитивни типове и променливи").

Областта, в която съществува и може да се използва една локална променлива, започва от реда, на който сме я декларирали и стига до затварящата фигурна скоба на тялото метода. Ако след нейното деклариране се опитаме да декларираме друга променлива с нейното име, например:

```
public static void main(String[] args) {
   int x = 3;
   int x = 4;
}
```

Компилаторът няма да ни позволи да направим това, със съобщение подобно на следното:

```
Duplicated variable <variable_name>.
```

В нашия случай **<variable_name>** е името на променливата **x**.

Блок (block) наричаме код, който се намира между фигурни скоби "**{**" и "**}**".

Ако декларираме променлива в блок, тя отново се нарича локална променлива, и областта й на съществуване е от реда, на който бъде декларирана, до затварящата скоба на блока.

Извикване на метод

Извикване на метод наричаме стартирането на изпълнението на кода, който е описан в тялото на метода.

Извикването на метода става просто като напишем името на метода **<method_name>**, следвано от кръглите скоби и накрая сложим знака за край на ред – ";":

```
<method_name>();
```

По-късно ще разгледаме и случая, когато извикваме метод, който има списък с параметри.

За да имаме ясна представа за извикването, ще покажем как бихме извикали метода, който използвахме в примерите по-горе – **printLogo()**:

```
printLogo();
```

Изходът от изпълнението на метода ще бъде:

```
Sun Microsystems
www.sun.com
```

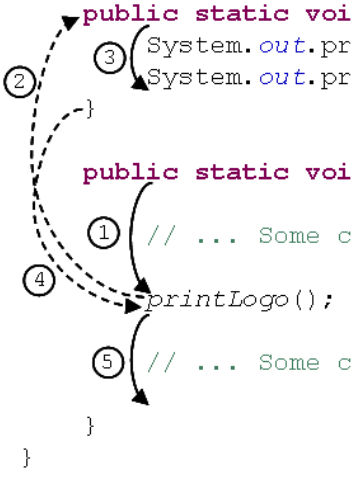
Предаване на контрола на програмата при извикване на метод

Когато изпълняваме един метод, той притежава контрола над програмата. Ако в тялото му обаче, извикаме друг метод, то тогава извикващия метод, ще предаде контрола на извиквания метод. След като извикваният метод приключи изпълнението си, той ще върне контрола на метода, който го е извикал. Изпълнението на първия метод ще продължи на следващия ред.

Например, нека от метода **main()** извикаме метода **printLogo()**. Първо ще се изпълни кодът от метода **main()**, който е означен с (1), след това контрола на програмата ще се предаде на метода **printLogo()** – пунктираната стрелка (2). След това, ще се изпълни кода в метода **printLogo()**, номериран с (3). След приключване на работата на метода **printLogo()** управлението на програмата ще бъде върнато на метода **main()** – пунктираната стрелка (4). Изпълнението на метода **main()** ще продължи от реда, който следва извикването на метода **printLogo()** – стрелката маркирана с (5):

MethodControlTest.java

```
public class MethodControlTest {

    public static void printLogo() {
        System.out.println("Sun Microsystems");
        System.out.println("www.sun.com");
    }

    public static void main(String[] args) {
        // ... Some code here ...
        printLogo();
        // ... Some code here ...
    }
}
```



От къде може да извикаме метод?

Един метод може да бъде извикван от следните места:

- От главния метод на програмата – **main()**:

```
public static void main(String[] args) {
   printLogo();
}
```

- От някой друг метод:

```
public static void printLogo() {
   System.out.println("Sun Microsystems");
   System.out.println("www.sun.com");
}

public static void printCompanyInformation() {

   // Invoking the printLogo() method
   printLogo();

   System.out.println("Address: Elm Str.");
}
```

- Методът може да бъде извикан в собственото си тяло. Това се нарича **рекурсия (recursion)**, но ще се запознаем по-подробно с нея в следващата глава – "Рекурсия".

Независимост между декларацията и извикването на метод

Когато пишем на Java, е позволено последователността, в която извикваме метода и го декларираме да е обърната, т.е. във файла, в който сме декларирали метода, извикването, да предхожда неговата декларация и имплементация.

За да стане по-нагледно, нека разгледаме следния пример:

```
public static void main(String[] args) {
   // ..
   printLogo();
   // ..
}

public static void printLogo() {
   System.out.println("Sun Microsystems");
   System.out.println("www.sun.com");
}
```

Ако създадем клас, който съдържа горния код, ще се убедим, че независимо че извикването на метода е на по-горен ред от декларацията на метода, програмата ще се компилира и изпълни без никакъв проблем. В някои други езици за програмиране, като например Паскал, извикването на метод, който е дефиниран по-надолу от мястото на извикването му, не е позволено.

Ако един метод бива извикван в същия клас, където е деклариран и имплементиран, то той може да бъде извикан на ред по-горен от реда на декларацията му.

Използване на параметри в методите

Много често, за да реши даден проблем, методът се нуждае от допълнителна информация, която зависи от контекста, в който той се изпълнява.

Например, ако имаме метод, който намира лице на квадрат, в тялото му е описан алгоритъма, по който се намира лицето (формулата: $S = a^2$). Въпреки това, при пресмятането на лицето на всеки отделен квадрат, методът ни ще се нуждае от дължината на страната му, за да изчисли конкретното лице спрямо нея. Затова, ние трябва да му я подадем някак.

Деклариране на метод

За да можем да подадем информация на даден метод, която е нужна за неговата работа, използваме **списък от параметри**. Този списък, поставяме между кръглите скоби в декларацията на метода, след името му:

```
public static <return_type> <method_name>(<parameters_list>) {
   // Method's body
}
```

Списъкът от параметри **<parameters_list>**, представлява списък от нула или повече **декларации на променливи**, разделени със запетая, които ще бъдат използвани в процеса на работа на метода:

```
<parameters_list> = [<type1> <name1>[, <typei> <namei>]],
където i = 2, 3,...
```

Когато създаваме метода и ни трябва дадена информация за реализирането на алгоритъма, избираме тази променлива от списъка от параметри, чийто тип **<typei>** ни е нужен и я използваме в метода съответно чрез името й **<namei>.**

Типът на параметрите в списъка може да бъде различен. Той може да бъде както примитивни типове – **int**, **double**, ... така и обекти (например **String** или масиви – **int[]**, **double[]**, **String[]**, ...).

Метод за извеждане фирмено лого – пример

За да добием по-ясна представа, нека модифицираме примера, който извежда логото на компанията "Sun Microsystems" по следния начин:

```
public static void printLogo(String logo) {
   System.out.println(logo);
}
```

По този начин, нашият метод вече няма да извежда само "Sun Microsystems", като резултат от изпълнението си, но логото на всяка компания, чието име подадем като параметър от тип **String**. В примера виждаме също как използваме информацията подадена ни в списъка от параметри – променливата **logo**, дефинирана в списъка от параметри, се използва в тялото на метода чрез името, с което сме я дефинирали.

Метод за сумиране цените на книги в книжарница – пример

По-горе казахме, че когато е нужно, можем да подаваме като параметри на метода и масиви – **int[]**, **double[]**, **String[]**, ... Нека в тази връзка разгледаме друг пример.

Ако сме в книжарница и искаме да пресметнем сумата, която дължим за всички книги, които желаем да закупим, можем да си създадем метод, който приема цените на отделните книги като масив от тип **double[]** и връща общата им стойност, която трябва да заплатим на продавача:

```
public static void printTotalAmountForBooks(double[] prices) {
```

```
   double totalAmount = 0;

   for (double singleBookPrice : prices) {
      totalAmount += singleBookPrice;
   }
   System.out.println("The total amount of all books is: " +
      totalAmount);
}
```

Поведение на метода в зависимост от входните данни

Когато декларираме метод с параметри, целта ни е всеки път, когато извикваме този метод, работата му да се променя в зависимост от входните данни. С други думи, алгоритъмът, който ще опишем в метода, ще бъде един, но крайният резултат ще бъде различен, в зависимост каква информация сме подали на метода чрез списъка от параметри.

Когато методът ни приема параметри, поведението му, зависи от тях.

Метод за извеждане знака на едно число - пример

За да стане ясно как поведението (изпълнението) на метода зависи от входните параметри, нека разгледаме следния метод, на който подаваме едно цяло число (от тип **int**), и в зависимост от това, дали числото е положително, отрицателно или нула, съответно той извежда в конзолата – "**Positive**", "**Negative**" или "**Zero**":

```
public static void printSign(int number) {
   if (number > 0) {
      System.out.println("Positive");
   } else if (number < 0) {
      System.out.println("Negative");
   } else {
      System.out.println("Zero");
   }
}
```

Списък с много параметри

До сега разглеждахме примери, в които методите имат списък от параметри, който се състои от един единствен параметър. Когато декларираме метод обаче, той може да бъде има толкова параметри, колкото са му необходими.

Например, когато търсим по-голямото от две числа, ние подаваме два параметъра:

```
public static void printMax(float number1, float number2) {
   float max = number1;
   if (number2 > number1) {
      max = number2;
   }
   System.out.println("Maximal number: " + max);
}
```

Особеност при декларацията на списък с много параметри

Когато в списъка с параметри декларираме повече от един параметър от един и същ тип, трябва да знаем, че не можем да използваме съкратения запис за деклариране на променливи от един и същ тип, както е позволено в самото тяло на метода:

```
float var1, var2;
```

Винаги трябва да указваме типа на параметъра в списъка с параметри на метода, независимо че някой от съседните му параметри е от същия тип.

Например, тази декларация на метод е неправилна:

```
public static void printMax(float var1, var2)
```

Съответно, правилният начин е:

```
public static void printMax(float var1, float var2)
```

Извикване на метод с параметри

Извикването на метод с много параметри става по същия начин, по който извиквахме на метод без параметри. Разликата е, че между кръглите скоби, след името на метода, поставяме стойности. Тези стойности ще бъдат присвоени на съответните параметри от декларацията на метода и при изпълнението си, метода ще работи с тях.

Ето няколко примера на извикване на методи с параметри:

```
printSign(-5);
printSign(balance);

printMax(100, 200);
```

Разлика между параметри и аргументи на метод

Преди да продължим, трябва да направим едно разграничение между наименованията на параметрите в списъка от параметри в декларацията на метода и стойностите, които подаваме при извикването на метода.

За по-голяма яснота, при деклариране на метода, елементите на списъка от параметрите му, ще наричаме **параметри** (някъде в литературата могат да се срещнат също като "формални параметри").

По време на извикване на метода, **стойностите**, които подаваме на метода, наричаме **аргументи** (някъде могат да се срещнат под понятието "фактически параметри").

С други думи, елементите на списъка от параметри **var1** и **var2**, наричаме параметри:

```
public static void printMax(float var1, float var2)
```

Съответно стойностите, при извикването на метода **-23.5** и **100**, наричаме аргументи:

```
printMax(100, -23.5);
```

Подаване на аргументи от примитивен тип

Както току-що научихме, когато в Java подадем като аргумент на метод дадена променлива, стойността ѝ се **копира** в параметъра от декларацията на метода. След това, копието ще бъде използвано в тялото на метода. Има, обаче, една особеност.

Когато съответният параметър от декларацията на метода е от **примитивен тип**, това практически не оказва никакво влияние на кода след извикването на метода.

Например, ако имаме следния метод:

```
public static void printNumber(int numberParam) {

   // Modifying the primitive-type parameter
   numberParam = 5;

   System.out.println("in printNumber() method, after the "
           + "modification, numberParam is: " + numberParam);
}
```

Извиквайки го от метода **main()**:

```
public static void main(String[] args) {
   int numberArg = 3;

   printNumber(numberArg);   // Copying the value 3 of the
                             // argument numberArg to the
                                // parameter numberParam
```

```
   System.out.println("in the main() method number is: " +
      numberArg);
}
```

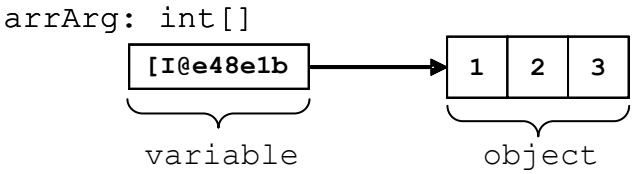
Стойността **3** на променливата **numberArg**, се копира в параметъра **numberParam**. След като бъде извикан методът **printNumber()**, на параметъра **numberParam** се присвоява стойността **5**. Това не рефлектира върху стойността на променливата **numberArg**, тъй като при извикването на метода, в **numberParam** се пази **копие** на стойността на подадения аргумент. Затова, методът **printNumber()** отпечатва числото **5**. Съответно, след извикването на метода **printNumber()**, в метода **main()** отпечатваме стойността на променливата **numberArg** и виждаме, че тя не е променена. Ето и изходът от изпълнението на горния код:

```
in printNumber() method, after the modification numberParam is:5
in the main() method number is: 3
```

Подаване на аргументи от референтен тип

Когато трябва да декларираме (и съответно извикаме) метод, чийто параметри са от **референтен тип** (например масиви), трябва да бъдем много внимателни.

Преди да обясним защо, нека припомним нещо от главата "Масиви". Масивът, като всеки референтен тип, се състои от променлива (референция) и стойност – реалната информация в паметта на компютъра (нека я наречем **обект**). Съответно в нашия случай обектът представлява реалният масив от елементи. Променливата пази адреса на обекта (елементите на масива) в паметта:



Когато оперираме с масиви, винаги го правим чрез променливата, с която сме ги декларирали. Така е и с всеки референтен тип. Следователно, когато подаваме аргумент от референтен тип, **стойността**, която е записана в променливата-аргумент, се **копира** в променливата, която е параметър в списъка от параметри на метода. Но какво става с обекта (реалния масив от елементи)? Копира ли се и той или не?

За да бъде по-нагледно обяснението, нека използваме следния пример: имаме метод **modifyArr()**, който модифицира първия елемент на подаден му като параметър масив, като го реинициализира със стойност **5** и след това отпечатва елементите на масива, оградени в квадратни скоби и разделени със запетайки:

```
public static void modifyArr(int[] arrParam) {
   arrParam[0] = 5;

   System.out.print("In modifyArr() the param is: ");
   System.out.println(Arrays.toString(arrParam));
}
```

Съответно, декларираме и метод **main()**, от който извикваме новосъздадения метод **modifyArr()**:

```
public static void main(String[] args) {
   int[] arrArg = new int[] { 1, 2, 3 };

   System.out.print("Before modifyArr() the argument is: ");
   System.out.println(Arrays.toString(arrArg));

   // Modifying the array's argument
   modifyArr(arrArg);

   System.out.print("After modifyArr() the argument is: ");
   System.out.println(Arrays.toString(arrArg));
}
```

Какъв ще е резултатът от изпълнението на този код? Нека погледнем:

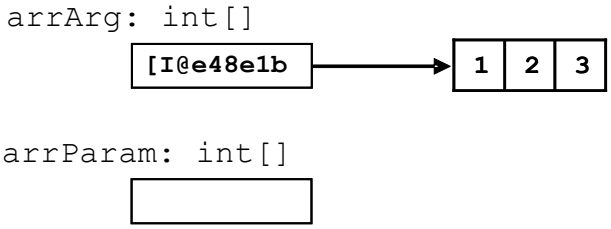
```
Before modifyArr() the argument is: [1, 2, 3]
In modifyArr() the param is: [5, 2, 3]
After modifyArr() the argument is: [5, 2, 3]
```

Забелязваме, че след изпълнението на метода **modifyArr()**, масивът към който променливата **arrArg** пази референция, не е **[1,2,3]**, а е **[5,2,3]**. Какво значи това?

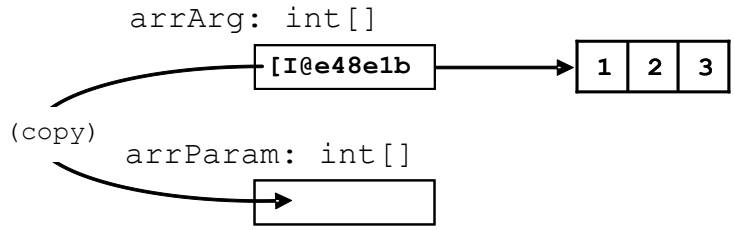
Причината за този резултат е, че при подаването на аргумент от референтен тип, се копира единствено стойността на променливата, която пази референция към обекта, но **не се прави копие на самия обект**.

При подаване на аргументи от референтен тип се копира само стойността на променливата, която пази референция към обекта в паметта, но не и самият обект.

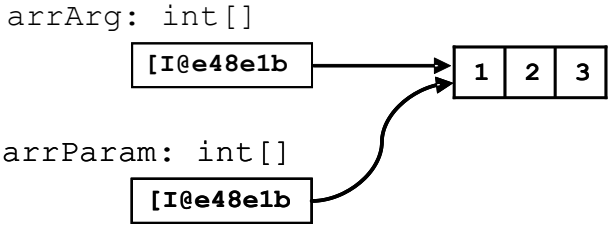
Нека онагледим казаното с няколко схеми, разглеждайки отново нашия пример. Преди извикването на метода **modifyArr()**, стойността на параметъра **arrParam** е неопределена и той не пази референция към никакъв конкретен обект (никакъв реален масив):



По време на извикването на **modifyArr()**, стойността, която е запазена в аргумента **arrArg**, се копира в параметъра **arrParam**:



По този начин, копирайки референцията към елементите на масива в паметта от аргумента в параметъра, ние указваме на параметъра да "сочи" към същия обект, към който "сочи" и аргументът:



И тъкмо това е моментът, за който трябва да сме внимателни, защото, ако извикания метод модифицира **обекта**, към който му е подадена референция, това може да повлияе на изпълнението на кода, който следва след изпълнението на метода (както видяхме в нашия пример – методът **printArr()** не отпечата масива, който бяхме очаквали).

Това е разликата между подаването на аргументи от примитивен и референтен тип.

Подаване на изрази като аргументи на метод

Когато извикваме метод, можем да подаваме цели изрази, като аргументи. Когато правим това, Java пресмята стойностите на тези изрази и по време на изпълнение (а когато е възможно и по време на компилация) заменя самия израз с пресметнатия резултат в извикването на метода. Например:

```
printSign(2 + 3);
```

```
float oldQuantity = 3;
float quantity = 2;
printMax(oldQuantity * 5, quantity * 2);
```

Съответно резултатът от изпълнението на тези методи е:

```
Positive
Maximal number: 15.0
```

Когато извикваме метод с параметри, трябва да спазваме някои определени правила, които ще обясним в следващите няколко подсекции.

Подаване на аргументи съвместими с типа на съответния параметър

Трябва да знаем, че можем да подаваме аргументи, които са съвместими по тип с типа, с който е деклариран съответния параметър в списъка от параметри на метода.

Например, ако параметърът, който методът очаква в декларацията си, е от тип **float**, при извикването на метода, може да подадем стойност, която е от тип **int**. Тя ще бъде преобразувана от компилатора до стойност от тип **float** и едва тогава ще бъде подадена на метода и той ще бъде изпълнен:

```
public static void printNumber(float number) {
   System.out.println("The float number is: " + number);
}

public static void main(String[] args) {
   printNumber(5);
}
```

В примера, при извикването на метода **printNumber()** в метода **main()**, първо целочисления литерал **5**, който по подразбиране е от тип **int**, се преобразува до съответната стойност с десетична запетая **5f**. Така преобразувана, тази стойност се подава на метода **printNumber()**.

Както предполагаме, изходът от изпълнението на този код е:

```
The float number is: 5.0
```

Съвместимост на стойността от израз и параметър на метод

Резултатът от пресмятането на някакъв израз, подаден като аргумент, трябва да е от същия тип, какъвто е типът на параметъра в декларацията на метода или от съвместим с него тип (вж. горната точка).

Например, ако се изисква параметър от тип **float**, е позволено стойността от пресмятането на израза да е например от тип **int**. Т.е. в горния пример,

ако вместо **printNumber(5)**, извикаме метода, като на мястото на **5**, поставим например израза **2+3**, резултатът от пресмятането на този израз, трябва да е от тип **float** (който метода очаква), или тип, който може да се преобразува до **float** безпроблемно (в нашия случай това е **int**). Нека леко модифицираме метода **main()** от предходната точка:

```
public static void main(String[] args) {
   printNumber(2 + 3);
}
```

Съответно, сумирането ще бъде извършено и целочисления резултат **5**, ще бъде преобразуван до еквивалента му с плаваща запетая **5f**, след което ще бъде извикан метода **printNumber()**. Резултатът отново ще бъде:

```
The float number is: 5.0
```

Спазване на последователността на типовете на аргументите

Стойностите, които се подават на метода при неговото извикване, трябва като типове, да са в същата последователност, в каквато са параметрите на метода при неговата декларация. Това е свързано със спецификацията (сигнатурата) на метода, за която говорихме по-горе.

За да стане по-ясно, нека разгледаме следния пример – нека имаме метод **printNameAndAge()**, който в декларацията си има списък от параметри, които са съответно от тип **String** и **int**, точно в тази последователност:

Person.java

```
public class Person {
   public static void printNameAndAge(String name, int age) {
      System.out.println(
         "I am " + name + ", " + age + " year(s) old.");
   }
}
```

Нека към нашия клас добавим метод **main()**, в който да извикаме нашия метод **printNameAndAge()**, като се опитаме да му подадем аргументи, които вместо **"Pesho"** и **25**, са в обратна последователност като типове – **25** и **"Pesho**:

```
public static void main(String[] args) {
   printNameAndAge(24, "Pesho"); // Wrong sequence of arguments
}
```

Компилаторът няма да намери метод, който се казва **printNameAndAge** и в същото време, има параметри, които са последователно от тип **int** и **String**. Затова, той ще ни уведоми за грешка:

```
The method printNameAndAge(int, String) in the type Person is not
applicable for the arguments (String, int)
```

Метод с променлив брой аргументи (var-args)

До момента, разглеждахме деклариране на методи, при което декларираме списък от параметри на метода, при който, когато извикваме нашия метод, аргументите, които подаваме трябва да са същият брой, какъвто е броят на параметрите в декларацията му.

Сега ще разгледаме деклариране на методи, която позволява **по време на извикване на метода**, броят на аргументите, които биват подавани, да е различен, в зависимост от нуждите на извикващия код.

Нека вземем примера, който разгледахме по-горе, в който пресмятаме сумата, която заплащаме на продавача в книжарницата след като сме си избрали книги. В него, като параметър на метода подавахме масив от тип **double[]**, в който се съхраняват цените на избраните от нас книги:

```
public static void printTotalAmountForBooks(double[] prices) {
   double totalAmount = 0;

   for (double singleBookPrice : prices) {
      totalAmount += singleBookPrice;
   }
   System.out.println("The total amount of all books is: " +
      totalAmount);
}
```

Така дефиниран, този метод предполага, че **винаги** преди да го извикаме, ще създадем масив с числа от тип **double** и ще го инициализираме с някакви стойности.

След създаването на Java 5.0, е възможно, когато трябва да подадем някакъв списък от стойности от **един и същ тип** на даден метод, вместо да го правим като подаваме масив, който съдържа тези стойности, да ги подадем на метода при извикването му, като аргументи, разделени със запетая.

Например, в нашия случай с книгите, вместо да създаваме масив, специално заради извикването на този метод:

```
double[] prices = new double[] { 3, 2.5 };
printTotalAmountForBooks(prices);
```

Можем директно да подадем списъка с цените на книгите, като аргументи на метода:

```
printTotalAmountForBooks(3, 2.5);
printTotalAmountForBooks(3, 5.1, 10, 4.5);
```

Този тип извикване на метода обаче е възможно само ако сме декларирали метода си, като метод, който приема променлив брой аргументи (var-args).

Деклариране на метод с параметър за променлив брой аргументи

Формално декларацията на метод с променлив брой аргументи е същата, каквато е декларацията на всеки един метод:

```
public static <return_type> <method_name>(<parameters_list>) {
   // Method's body
}
```

Разликата е, че **<parameters_list>** се декларира по следния начин:

```
<parameters_list> =
   [<type1> <name1>[, <typei> <namei>], <va_type>... <va_name>]
където i= 2, 3, ...
```

Последният елемент от декларацията на списъка – **<va_name>**, е този, който позволява подаването на произволен брой аргументи от типа **<va_type>**, при всяко извикване на метода.

При декларацията на този елемент, след типа му, които трябва да подадем – **<va_type>**, трябва да добавим три точки: "**<va_type>...**". По подобие на останалите типове на параметри в списъка от параметри на метода, **<va_type>** може да бъде както примитивен тип, така и референтен.

Правилата и особеностите за останалите елементи от списъка с параметри на метода, предхождащи var-args параметъра **<va_name>**, са същите, каквито ги разгледахме по-горе в тази глава.

За да стане по-ясно казаното до тук, нека разгледаме още един пример:

```
public static long calcSum(int ... elements) {
   long sum = 0;
   for (int element : elements) {
      sum += element;
   }
   return sum;
}

public static void main(String[] args) {
```

```
   long sum = calcSum(2, 5);
   System.out.println(sum);

   long sum2 = calcSum(4, 0, -2, 12);
   System.out.println(sum2);

   long sum3 = calcSum();
   System.out.println(sum3);
}
```

Примерът сумира числа, като техният брой не е предварително известен. Методът може да бъде извикан с един, два или повече параметъра, а също и без параметри. Ако изпълним примера, ще получим следния резултат:

```
7
14
0
```

Същност на декларацията на параметър за променлив брой аргументи

Параметърът от формалната дефиниция по-горе, който позволява подаването на променлив брой аргументи при извикването на метода – **<va_name>**, всъщност е име на **масив** от тип **<va_type>**. При извикването на метода, аргументите от тип **<va_type>** или тип съвместим с него, които подаваме на метода, (независимо от броя им) ще бъдат съхранени в този масив. След това те ще бъдат използвани в тялото на метода. Достъпът и работата до тези елементи става по тривиалния начин, по който работим с масиви.

За да стане по-ясно, нека преработим метода, който пресмята сумата на избраните от нас книги, да приема произволен брой аргументи:

```
public static void printTotalAmountForBooks(double... prices) {
   double totalAmount = 0;

   for (double singleBookPrice : prices) {
      totalAmount += singleBookPrice;
   }
   System.out.println("The total amount of all books is: " +
      totalAmount);
}
```

Виждаме, че единствената промяна бе да сменим декларацията на масива **prices** да бъде **double...**, а не **double[]**. Въпреки това, в тялото на нашия метод, **prices** отново е масив от тип **double[]**, който използваме по познатия ни начин в тялото на метода.

Сега можем да извикаме нашия метод, без да декларираме предварително масив от числа, който да подаваме като аргумент на метода:

```
public static void main(String[] args) {

   printTotalAmountForBooks(3, 2.5);

   printTotalAmountForBooks(1, 2, 3.5, 7.5);
}
```

Съответно резултатът от двете извиквания на метода ще бъде:

```
The total amount of all books is: 5.5
The total amount of all books is: 14.0
```

Както вече се досещаме, тъй като сам по себе си **prices** е масив, можем да декларираме и инициализираме масив преди извикването на нашия метод и да го подадем този масив като стойност:

```
public static void main(String[] args) {

   double[] prices = new double[] { 3, 2.5 };

   // Passing initialized array as var-arg:
   printTotalAmountForBooks(prices);
}
```

Това е напълно легално и резултатът от изпълнението на този код ще е следният:

```
The total amount of all books is: 5.5
```

Позиция на декларацията на параметъра за променлив брой аргументи

Един метод, който може да приема произволен брой аргументи, **може да има и други параметри** в списъка си от параметри.

Например, следният метод, приема като първи параметър елемент от тип **String**, а след това нула или повече елементи от тип **int**:

```
public static void doSth(String strParam, int... x) { }
```

Особеното, на което трябва да обърнем внимание е, че елементът от списъка от параметри в дефиницията на метода, който позволява подаването на произволен брой аргументи, независимо от броя на останалите параметри, трябва да е **винаги** на последно място.

Елементът от списъка от параметри на един метод, който позволява подаването на произволен брой аргументи при извикването на метода, трябва да се декларира винаги на последно място в списъка от параметри на метода.

Ако се опитаме да поставим декларацията на var-args параметъра x, от последния пример, да не бъде на последно място в списъка от параметри на метода:

```
public static void doSth(int... x, String strParam) { }
```

Компилаторът ще изведе следното съобщение за грешка:

```
The variable argument type int of the method doSth must be the last
parameter
```

Ограничение на броя на параметрите за променлив брой аргументи

Също така, трябва да знаем, че в декларацията на един метод не може да имаме повече от един параметър, който позволява подаването на променлив брой аргументи при извикването на метода. С други думи, ако се опитаме да компилираме следната декларация на метод:

```
public static void doSth(int... x, String... z) {}
```

Компилаторът ще изведе отново познатото съобщение за грешка:

```
The variable argument type int of the method doSth must be the last
parameter
```

Това правило е частен случай на правилото за позицията на var-args параметъра – да бъде на последно място в списъка от параметри.

Особеност при извикване на метод с променлив брой параметри, без подаване на нито един параметър

След като се запознахме с декларацията и извикването на методи с променлив брой аргументи и разбрахме същността им, може би възниква въпроса, какво ще стане, ако не подадем нито един елемент на такъв метод по време на извикването му?

Например, какъв ще е резултатът от изпълнението на нашия метод за пресмятане цената на избраните от нас книги, в случая, когато не сме си харесали нито една книга:

```
public static void main(String[] args) {
```

```
   printTotalAmountForBooks();
}
```

Виждаме, че компилацията на този код минава без проблеми и след изпълнението резултатът е следният:

```
The total amount of all books is: 0.0
```

Това е така, защото, въпреки че не сме подали нито една стойност на нашия метод, при извикването на метода, масивът **double... prices** е създаден, но няма нито един елемент.

Това е добре да бъде запомнено, тъй като дори да няма подадени стойности, Java се грижи да **инициализира масива**, в който се съхраняват променливия брой аргументи.

Метод променлив брой параметри – пример

Имайки предвид как дефинираме методи с променлив брой аргументи, можем да запишем добре познатият ни **main()** метод по следния начин:

```
public static void main(String... args) {
   // Method body comes here
}
```

Горната дефиниция е напълно валидна и се приема от компилатора.

Варианти на методи (method overloading)

Когато декларираме един метод, чието име съвпада с името на друг метод, но сигнатурите на двата метода се различават по **списъка от параметри** (броя на елементите в него или подредбата им), казваме, че имаме различни **варианти на този метод** (**method overloading**).

Например, ако си представим, че имаме задача да напишем програма, която изографисва букви и цифри. Съответно можем да си представим, че нашата програма, може да има методите за изографисване съответно на низове **drawString(String string)**, цели числа – **drawInt(int number)**, десетични числа – **drawFloat(float number)** и т.н.:

```
public static void drawString(String string) {
   // Draw string
}

public static void drawInt(int number) {
   // Draw integer
}
```

```
public static void drawFloat(float number) {
   // Draw float number
}
```

Но също така можем да си създадем съответно само варианти на един метод – **draw(…)**, който приема различни типове параметри, в зависимост от това, какво искаме изографисаме:

```
public static void draw(String str) {
   // Draw string
}

public static void draw(int number) {
   // Draw integer
}

public static void draw(float number) {
   // Draw float number
}
```

Значение на параметрите в сигнатурата на метода

Както казахме по-горе, за спецификацията (сигнатурата) на един метод, в Java, единствените елементи от списъка с параметри, които имат значение, са **типовете на параметрите** и **последователността, в която са изброени**. Имената на параметрите нямат значение за еднозначното деклариране на метода.

За еднозначното деклариране на метод в Java, по отношение на списъка с параметри на метода, единствено имат значение:

- **типът на параметъра**
- **последователността на типовете в списъка от параметри**

Имената на параметрите не се вземат под внимание.

Например за Java, следните две декларации, са декларации на един и същ метод, тъй като типовете на параметрите в списъка от параметри са едни и същи – **int** и **float**, независимо от имената на променливите, които сме поставили – **param1** и **param2** или **arg1** и **arg2**:

```
public static void doSomething(int param1, float param2)
public static void doSomething(int arg1, float arg2)
```

Ако въпреки всичко, декларираме два метода в един и същ клас, по този начин, компилаторът ще изведе съобщение за грешка, подобно на следното:

```
Duplicate method doSomething(int, float) in type <the_name_of_
your_class>
```

Където **<the_name_of_your_class>** е името на класа, в който се опитваме да декларираме методите.

Ако обаче в примера, който разгледахме, някои от **параметрите на една и съща позиция в списъка от параметри са от различен тип**, тогава за Java, това са два напълно различни метода, или по-точно, **варианти на един метод с даденото име**.

Например, ако във втория метод, вторият параметър от списъка на единия от методите – **float arg2**, го декларираме да не бъде от тип **float**, а **int**, тогава това са два различни метода с различна спецификация – **doSomething(int, float)** и **doSomething(int, int)**. Вторият елемент от сигнатурата им – **списъкът от параметри**, е напълно различен, тъй като типовете на вторите им елементи от списъка са различни:

```
public static void doSomething(int arg1, float arg2)

public static void doSomething(int param1, int param2)
```

В този случай, дори да поставим едни и същи имена на параметрите в списъка, компилаторът ще ги приеме, тъй като за него това са различни методи:

```
public static void doSomething(int param1, float param2)

public static void doSomething(int param1, int param2)
```

Компилаторът отново "няма възражения", ако отново декларираме вариант на метод, но този път вместо да подменяме типа на втория параметър, просто разменим местата на параметрите на втория метод:

```
public static void doSomething(int param1, float param2)

public static void doSomething(float param2, int param1)
```

Тъй като **последователността на типовете** на параметрите в списъка с параметри е различна, съответно и спецификациите на методите са различни. Щом списъците с параметри са различни, еднаквите имена (**doSomething**) нямат отношение към еднозначното деклариране на методите в нашия клас – имаме различни сигнатури.

Триъгълници с различен размер – пример

Искаме да напишем програма, която отпечатва триъгълници, като тези, показани по-долу:

```
                                        1
          1                             1 2
          1 2                           1 2 3
          1 2 3                       1 2 3 4
          1 2 3 4                       1 2 3 4 5
n=5   ->  1 2 3 4 5       n=6   ->      1 2 3 4 5 6
          1 2 3 4                       1 2 3 4 5
          1 2 3                       1 2 3 4
          1 2                           1 2 3
          1                             1 2
                                        1
```

Нека разгледаме едно възможно решение и обясним как работи то:

Triangle.java

```
import java.util.Scanner;

public class Triangle {

   public static void main(String[] args) {

      // Entering the value of the variable n
      System.out.print("n = ");
      Scanner input = new Scanner(System.in);
      int n = input.nextInt();
      System.out.println();

      // Printing the upper part of the triangle
      for (int line = 1; line <= n; line++) {
         printLine(1, line);
      }

      // Printing the bottom part of the triangle
      // that is under the longest line
      for (int line = n - 1; line >= 1; line--) {
         printLine(1, line);
      }
   }

   private static void printLine(int start, int end) {
      for (int i = start; i <= end; i++) {
         System.out.print(" " + i);
```

```
        }
        System.out.println();
    }
}
```

Тъй като, можем да печатаме в конзолата ред по ред, разглеждаме триъгълниците, като съставени от редове (а не от колони). Следователно, за да ги изведем в конзолата, трябва да имаме средство, което извежда редовете на триъгълниците. За целта, създаваме метода **printLine()**.

В него, с помощта на цикъл **for**, отпечатваме в конзолата редица от последователни числа. Първото число от тази редица е първия параметър в списъка от параметри на метода – променливата **start**. Последният елемент на редицата е числото, подадено на метода, като втори параметър (именуван с **end**) в списъка с параметри.

Забелязваме, че тъй като числата са последователни, дължината (броя числа) на всеки ред, съответства на разликата между втория параметър **end** и първия – **start**, от списъка с параметри на метода (това ще ни послужи малко по-късно, когато конструираме триъгълниците).

След това създаваме алгоритъмът за отпечатването на триъгълниците, като цялостни фигури, в метода **main()**. Чрез класа **Scanner** въвеждаме стойността на променливата **n** и извеждаме празен ред.

След това, в два последователни **for**-цикъла конструираме триъгълника, който трябва да се изведе, за даденото **n**. В първия цикъл отпечатваме последователно всички редове от горната част на триъгълника до средния – най-дълъг ред, включително. Във втория цикъл, отпечатваме редовете на триъгълника, които трябва да се изведат под средния – най-дълъг ред.

Както отбелязахме по-горе, номерът на реда, съответства на броя на елементи (числа) намиращи се на съответния ред. И тъй като винаги започваме от числото **1**, номерът на реда, в горната част от триъгълника, винаги ще е равен на последния елемент на редицата, която трябва да се отпечата на дадения ред. Следователно, можем да използваме това при извикването на метода **printLine()**, тъй като той изисква точно тези параметри за изпълнението на задачата си.

Също ни прави впечатление, че броят на елементите на редиците, се увеличава с единица и съответно, последният елемент на всяка по-долна редица, трябва да е с единица по-голям от последния елемент на редицата от предходния ред. Затова, при всяко "завъртане" на първия for-цикъл, подаваме на метода **printLine()**, като първи параметър **1**, а като втори – текущата стойност на променливата **line**. Тъй като при всяко изпълнение на тялото на цикъла **line** се увеличава с единица, на при всяка итерация методът **printLine()** ще отпечатва редица с един елемент повече от предходния ред.

При втория цикъл, който отпечатва долната част на триъгълника, следваме обратната логика. Колкото по-надолу печатаме, редиците трябва да се смаляват с по един елемент и съответно, последния елемент на всяка редица, трябва да са с единица по-малък от последния елемент на редицата от предходния ред. От тук задаваме началното условие за стойността на променливата line във втория цикъл: **line=n-1**. След всяко завъртане на цикъла намаляваме стойността на **line** с единица и я подаваме като втори параметър на **printLine()**.

Една оптимизация, която можем да направим е да изнесем логиката, която отпечатва един триъгълник в отделен метод. Забелязваме, че логически, печатането на триъгълник е ясно обособено, затова можем да декларираме метод с един параметър (стойността, която въвеждаме от клавиатурата) и да го извикаме в метода **main()**:

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.print("n = ");
   int n = input.nextInt();
   input.close();
   System.out.println();

   printTriangle(n);
}

private static void printTriangle(int n) {

   // Printing the upper part of the triangle
   for (int line = 1; line <= n; line++) {
      printLine(1, line);
   }

   // Printing the bottom part of the triangle
   // that is under the longest line
   for (int line = n - 1; line >= 1; line--) {
      printLine(1, line);
   }
}
```

Ако изпълним програмата и въведем за **n** стойност 3, ще получим следния резултат:

```
n = 3

 1
 1 2
 1 2 3
 1 2
```

```
1
```

Разстояние между два месеца – пример

Да разгледаме следната задача: искаме да напишем програма, която при зададени две числа, които трябва да са между **1** и **12**, за да съответстват на номер на месец от годината, да извежда броя месеци, които делят тези два месеца. Съобщението, което програмата трябва да отпечатва в конзолата трябва да е "**There is X months period from Y to Z.**", където **X** е броят на месеците, който трябва да изчислим, а **Y** и **Z**, са съответно имената на месеците за начало и край на периода.

Прочитаме задачата внимателно и се опитваме да я разбием на подпроблеми, които да решим лесно и след това интегрирайки решенията им да получим решението на цялата задача. Виждаме, че трябва да решим следните подзадачки:

- Да въведем номерата на месеците за начало и край на периода.
- Да пресметнем периода между въведените месеци.
- Да изведем съобщението.
- В съобщението вместо числата, които сме въвели за начален и краен месец на периода, да изведем съответстващите им имена на месеци на английски.

Ето едно възможно решение е следното:

Months.java

```
import java.util.Scanner;

public class Months {
   public static void sayMonth(int month) {
      String monthName = "";
      switch (month) {
      case 1:
         monthName = "January";
         break;
      case 2:
         monthName = "February";
         break;
      case 3:
         monthName = "March";
         break;
      case 4:
         monthName = "April";
         break;
```

```
      case 5:
         monthName = "May";
         break;
      case 6:
         monthName = "June";
         break;
      case 7:
         monthName = "July";
         break;
      case 8:
         monthName = "August";
         break;
      case 9:
         monthName = "September";
         break;
      case 10:
         monthName = "October";
         break;
      case 11:
         monthName = "November";
         break;
      case 12:
         monthName = "December";
         break;
      default:
         System.out.println("Error!");
         break;
      }
      System.out.print(monthName);
   }

   public static void sayPeriod(int startMonth, int endMonth) {
      int period = endMonth - startMonth;
      if (period < 0) {
         // Fix negative distance
         period = period + 12;
      }

      System.out.printf(
         "There is %d months period from ", period);
      sayMonth(startMonth);
      System.out.print(" to ");
      sayMonth(endMonth);
      System.out.println(".");
   }

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
```

```
        System.out.print("First month (1-12): ");
        int firstMonth = input.nextInt();

        System.out.print("Second month (1-12): ");
        int secondMonth = input.nextInt();

        sayPeriod(firstMonth, secondMonth);
        input.close();
    }
}
```

Решението на първата подзадача е тривиално. В метода **main()** използваме класа **Scanner** и получаваме номерата на месеците за периода, чиято дължина търсим.

След това забелязваме, че пресмятането на периода и отпечатването на съобщението може да се обособи логически като подзадачка, и затова създаваме метод **sayPeriod()** със списък от два параметъра – числа, съответстващи на номерата на месеците за начало и край на периода. Той няма да връща стойност, но ще пресмята периода и ще отпечатва съобщението описано в условието на задачата с помощта на стандартния изход – **System.out**.

Очевидното решение, за намирането на дължината на периода между два месеца, е като извадим поредния номер на началния месец от този на месеца за край на периода. Съобразяваме обаче, че ако номера на втория месец е по-малък от този на първия, тогава потребителят е имал в предвид, че вторият месец, не се намира в текущата година, а в следващата. Затова, ако разликата между двата месеца е отрицателна, към нея добавяме **12** – дължината на една година в брой месеци, и получаваме дължината на търсения период. След това извеждаме съобщението, като за отпечатването на имената на месеците, чийто пореден номер получаваме от потребителя, използваме метода **sayMonth()**.

За имплементацията на този метод се досещаме, че най-удачното решение ще бъде да използваме условната конструкция **switch-case**, с която да съпоставим на всяко число, съответстващото му име на месец от годината. Ако стойността на входния параметър не е някоя между стойностите **1** и **12,** извеждаме "**Error!**".

След това извикваме метода **sayPeriod()** в метода **main()**, подавайки му въведените от потребителя числа за начало и край на периода и с това сме решили задачата.

Ето какъв би могъл да е изходът от програмата при входни данни 2 и 6:

```
First month (1-12): 2
Second month (1-12): 6
```

```
There is 4 months period from February to June.
```

Връщане на резултат от метод

До момента, винаги давахме примери, в които методът извършва някакво действие, евентуално отпечатва нещо в конзолата, приключва работата си и с това се изчерпват "задълженията" му. Истината обаче е, че един метод, освен просто да изпълнява списък от действия, когато ни е нужно, може да върне някакъв резултат от дейността си.

Деклариране на метод с връщана стойност

Ако погледнем отново как декларираме метод:

```
public static <return_type> <method_name>(<parameters_list>)
```

Ще си припомним, че когато обяснявахме за това, казахме, че на мястото на **<return type>** поставяме **void**. Сега ще разширим дефиницията, като кажем, че на това място може да стои не само **void**, но и произволен тип – примитивен (**int**, **float**, **double**, …) или референтен (например **String** или масив), в зависимост от това, какъв тип е резултатът от изпълнението на метода.

Например, ако вземем примера с метод, който изчислява лице на квадрат, вместо да отпечатваме стойността в конзолата, методът може да я върне като резултат. Ето как би изглеждала декларацията на метода:

```
public static double calcSquareSurface(double sideLength)
```

Виждаме, че резултатът от пресмятането на лицето е от тип **double**.

Употреба на връщаната стойност

Когато методът бъде изпълнен и върне стойност, можем да си представяме, че Java поставя тази стойност на мястото, където е било извикването на метода и продължава работа с нея. Съответно, тази върната стойност, можем да използваме от извикващия метод с различни цели.

Присвояване на променлива

Може да присвоим резултата от изпълнението на метода, на променлива от подходящ тип:

```
// getCompanyLogo() returns a string
String companyLogo = getCompanyLogo();
```

Употреба в изрази

След като един метод върне резултат, този резултат, може да го използваме в изрази.

Например, за да намерим общата цена трябва да получим единичната такава и да умножим по количеството:

```
float totalPrice = getSinglePrice() * quantity;
```

Подаване като стойност в списък от параметри на друг метод

Можем да подадем резултата от работата на един метод, като стойност в списъка от параметри на друг метод:

```
System.out.println(getCompanyLogo());
```

В този пример, отначало извикваме метода **getCompanyLogo()**, подавайки го като аргумент на метода **println()**. След като методът **getCompanyLogo()** бъде изпълнен, той ще върне резултат, например – "**Sun Microsystems**". Тогава Java ще "подмени" извикването на метода, с резултата, който е върнат от изпълнението му и можем да приемем, че в кода имаме:

```
System.out.println("Sun Microsystems");
```

Тип на връщаната стойност

Както казахме малко по-рано, резултатът, който връща един метод може да е от всякакъв тип – **int**, **String**, масив и т.н. Когато обаче, като тип на връщаната стойност бъде употребена ключовата дума **void**, с това означаваме, че методът не връща никаква стойност.

Операторът return

За да накараме един метод да връща стойност, трябва в тялото му, да използваме ключовата дума **return**, следвана от резултата на метода:

```
public static <return_type> <method_name>(<parameters_list>) {
   // Some code that is preparing the method's result comes here
   return <method's_result>;
}
```

Съответно **<method's_result>**, е от тип **<return_type>**. Например:

```
public static int multiply(int number1, int number2) {
   int result = number1 * number2;
   return result;
```

```
}
```

В този метод, след умножението, благодарение на **return**, методът ще върне резултата от изпълнението на метода – целочислената променлива **result**.

Резултат от тип, съвместим, с типа на връщаната стойност

Резултатът, който се връща от метода, може да е от тип, който е **съвместим** (който може неявно да се преобразува) с типа на връщаната стойност **<return_type>**.

Например, може да модифицираме последния пример, в който типа на връщаната стойност да е от тип **float**, а не **int** и запазим останалия код:

```
public static float multiply(int number1, int number2) {
   int result = number1 * number2;

   // Behind the scene, Java executes for us the cast:
   //   return ((float) result);
   return result;
}
```

В този случай, след изпълнението на умножението, резултатът ще е от тип **int**. Въпреки това, на реда, на който връщаме стойността, той ще бъде неявно преобразуван до цяло число от тип **float** и едва тогава, ще бъде върнат като резултат.

Поставяне на израз след оператора return

Позволено е, когато това няма да направи кода нечетим, след ключовата дума **return**, да поставяме директно изрази:

```
public static int multiply(int number1, int number2) {
   return number1 * number2;
}
```

В тази ситуация, след като изразът **number1 * number2** бъде изчислен, резултатът от него ще бъде заместен на мястото на израза и ще бъде върнат от оператора **return**.

Характеристики на оператора return

При изпълнението си операторът **return** извършва две неща:

- **Прекратява** изпълнението на метода.
- Връща резултата от изпълнението на метода на извикващия метод.

Във връзка с първата характеристика на оператора **return**, трябва да кажем, че тъй като той прекратява изпълнението на метода, след него до затварящата скоба, не трябва да има други оператори.

Ако все пак направим това, компилаторът няма да ни позволи да продължим компилирането, докато **return** не остане последен оператор в тялото на метода:

```
public static int add(int number1, int number2) {
   int result = number1 + number2;
   return result;

   // Let us try to "clean" the result variable here:
   result = 0;
}
```

В този случай компилацията ще е неуспешна. За редовете след **return**, компилаторът ще изведе съобщение за грешка, подобно на следното:

```
Unreachable code
```

Когато методът има тип на връщана стойност **void**, тогава след **return**, не трябва да има израз, който да бъде върнат. В този случай употребата на **return** е единствено за прекратяване на метода:

```
public void printPositiveNumber(int number) {
   if (number <= 0) {
      // If the number is NOT positive, terminate the method
      return;
   }
   System.out.println(number);
}
```

Последното, което трябва да научим за оператора **return** е, че може да бъде извикван от няколко места в метода, като е гарантирано, че всеки следващ оператор **return** е достъпен при определени входни условия.

Нека разгледаме примера за метод, който получава като параметри две числа и в зависимост дали първото е по-голямо от второто, двете са равни, или второто е равно на първото, връща съответно 1,0 и -1:

```
public int compareTo(int number1, int number2) {
   if (number1 > number2) {
      return 1;
   } else if (number1 == number2) {
      return 0;
   } else {
      return -1;
```

```
    }
}
```

Защо типът на връщаната стойност не е част от сигнатурата на метода?

В Java не е позволено да имаме няколко метода, които имат еднакви параметри, но различен тип на връщаната стойност. Това означава, че следния код няма да се компилира:

```
public static int add(int number1, int number2) {
    return (number1 + number2);
}

public static double add(int number1, int number2) {
    return (number1 + number2);
}
```

Причината за това ограничение е, че компилаторът не знае кой от двата метода да извика и няма как да разбере. Затова, още при опита за декларация на двата метода, той ще изведе следното съобщение за грешка:

```
Duplicate method add(int, int) in type <the_name_of_your_class>
```

където **<the_name_of_your_class>** е името на класа, в който се опитваме да декларираме двата метода.

Преминаване от Фаренхайт към Целзий – пример

В тази задача от нас се иска да напишем програма, която при подадена от потребителя телесна температура, измерена в градуси Фаренхайт, я преобразува и извежда в съответстващата температура в градуси Целзий със следното съобщение: "**Your body temperature in Celsius degrees is X**", където **X** е съответно градусите Целзий. Също така, ако измерената температура в градуси Целзий е по-висока от 37 градуса, нашата програма, трябва да предупреждава потребителя, че е болен, със съобщението "**You are ill!**".

Правим бързо проучване в Интернет и разбираме, че формулата, която ни трябва за преобразуването е **ºC = (ºF - 32) * 5 / 9**, където съответно с **ºC** отбелязваме температурата в градуси Целзий, а с **ºF** – съответно тази в градуси Фаренхайт.

Анализираме поставената задача и виждаме, че подзадачките, на които може да се раздели са следните:

- Вземаме температурата измервана в градуси Фаренхайт като вход от клавиатурата (потребителят ще трябва да я въведе).

- Преобразуваме полученото число в съответното му число за температурата измервана в градуси Целзий.
- Извеждаме съобщение за преобразуваната температура в Целзий.
- Ако температурата е по-висока от 37 ºC, извеждаме съобщение на потребителя, че той е болен.

Ето едно примерно решение:

TemperatureConverter.java

```
import java.util.Scanner;

public class TemperatureConverter {
   public static double convertFahrenheitToCelsius(
         double temperatureF) {
      double temperatureC = (temperatureF - 32) * 5 / 9;
      return temperatureC;
   }

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println(
         "Enter your body temperature in Fahrenheit degrees: ");
      double temperature = input.nextDouble();

      temperature =
         convertFahrenheitToCelsius(temperature);

      System.out.printf(
         "Your body temperature in Celsius degrees is %f.%n",
            temperature);

      if (temperature >= 37) {
         System.out.println("You are ill!");
      }

      input.close();
   }
}
```

Операциите по въвеждането на температурата и извеждането на съобщенията са тривиални, и за момента прескачаме решението им, като се съсредоточаваме върху преобразуването на температурите. Виждаме, че това е логически обособено действие, което може да изведем в отделен метод. Това, освен, че ще направи кода ни по-четим, ще ни даде възможност в бъдеще, ако ни се наложи да правим подобно преобразование отново, да преизползваме този метод. Декларираме метода

convertFahrenheitToCelsius(), със списък от един параметър с името **temperatureF**, който представлява измерената температура в градуси Фаренхайт и връща съответно число от тип **double**, което представлява преобразуваната температура в Целзий. В тялото му описваме откритата в Интернет формула чрез синтаксиса на Java.

След като сме приключили с тази стъпка от решението на задачата, решаваме, че останалите стъпки няма нужда да ги извеждаме в методи, а е достатъчно да ги имплементираме в метода **main()** на класа.

С помощта на класа **Scanner**, получаваме телесната температура на потребителя, като предварително сме го попитали за нея със съобщението "**Enter your body temperature in Fahrenheit degrees**".

След това извикваме метода **convertFahrenheitToCelsius()**, резултатът от който съхраняваме в променливата **temperature**.

С помощта на метода **printf()** на **System.out**, извеждаме съобщението "**Your body temperature in Celsius degrees is X**", където **X** го заменяме със стойността на **temperature**.

Последната стъпка, която трябва да се направи е с условната конструкция **if**, да проверим дали температурата е по-голяма или равна на 37 градуса Целзий и ако е, да изведем съобщението, че потребителят е болен.

Ето примерен изход от програмата:

```
Enter your body temperature in Fahrenheit degrees:
100
Your body temperature in Celsius degrees is 37,777778.
You are ill!
```

Валидация на данни – пример

В тази задача, трябва да напишем програма, която пита потребителя колко е часът (с извеждане на въпроса "**What time is it?**"). След това потребителят, трябва да въведе две числа, съответно за час и минути. Ако въведените данни представляват валидно време, програмата, трябва да изведе съобщението "**The time is HH:mm now.**", където с **HH** съответно сме означили часа, а с **mm** – минутите. Ако въведените час или минути не са валидни, програмата трябва да изведе съобщението "**Incorrect time!**".

След като прочитаме условието на задачата внимателно, стигаме до извода, че решението на задачата може да се разбие на следните подзадачи:

- Получаване на входа за час и минути.
- Проверка на валидността на входните данни.
- Извеждаме съобщение за грешка или валидно време.

Знаем, че обработката на входа и извеждането на изхода няма да бъдат проблем за нас, затова решаваме да разрешим проблема с валидността на входните данни, т.е. валидността на числата за часове и минути. Знаем, че часовете варират от 0 до 23 включително, а минутите съответно от 0 до 59 включително. Тъй като данните (часове и минути) не са еднородни решаваме да създадем два отделни метода, единият от които проверява валидността на часовете, а другия – на минутите.

Ето едно примерно решение:

DataValidation.java

```
import java.util.Scanner;

public class DataValidation {

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("What time is it?");

      System.out.print("Hours: ");
      int hours = input.nextInt();

      System.out.print("Minutes: ");
      int minutes = input.nextInt();

      boolean isValidTime =
         validateHours(hours) && validateMinutes(minutes);
      if (isValidTime) {
         System.out.printf(
            "The time is %d:%d now.%n", hours, minutes);
      } else {
         System.out.println("Incorrect time!");
      }
   }

   public static boolean validateHours(int hours) {
      boolean result = (hours >= 0) && (hours < 24);
      return result;
   }

   public static boolean validateMinutes(int minutes) {
      boolean result = (minutes >= 0) && (minutes <= 59);
      return result;
   }
}
```

Методът, който проверява часовете, го кръщаваме **validateHours()**, като той приема едно число от тип **int**, за часовете и връща резултат от тип

boolean, т.е. **true** ако въведеното число е валиден час и **false** в противен случай:

```
public static boolean validateHours(int hours) {
   boolean result = (hours >= 0) && (hours < 24);
   return result;
}
```

По подобен начин, декларираме метод, който проверява валидността на минутите. Наричаме го **validateMinutes()**, като приема като списък от един параметър цяло число, за минути и има тип на връщана стойност – **boolean**. Ако въведеното число удовлетворява условието, което описахме по-горе, да е между 0 и 59 включително, методът ще върне като резултат **true**, иначе – **false**:

```
public static boolean validateMinutes(int minutes) {
   boolean result = (minutes >= 0) && (minutes <= 59);
   return result;
}
```

След като сме готови с най-сложната част от задачата, декларираме метода **main()**. В тялото му, извеждаме въпроса, който беше указан в условието на задачата – "**What time is it?**". След това с помощта на класа **Scanner**, вземаме от потребителя числата за часове и минути, като резултатите ги съхраняваме в целочислените променливи, съответно **hours** и **minutes**:

```
Scanner input = new Scanner(System.in);
System.out.println("What tidme is it?");

System.out.print("Hours: ");
int hours = input.nextInt();

System.out.print("Minutes: ");
int minutes = input.nextInt();
```

Съответно, резултата от валидацията го съхраняваме в променлива от тип **boolean** – **isValidTime**, като последователно извикваме методите, които вече декларирахме - **validateHours()** и **validateMinutes()**, като съответно им подаваме като аргументи променливите **hours** и **minutes**. За да ги валидираме едновременно, обединяваме резултатите от извикването на методите с оператора за логическо "и" – **&&**:

```
boolean isValidTime =
   validateHours(hours) && validateMinutes(minutes);
```

След като сме съхранили резултата, дали въведеното време е валидно или не, в променливата **isValidTime**, го използваме в условната конструкция **if**,

за да изпълним и последния подпроблем от цялостната задача – извеждането на информация към потребителя дали времето, въведено от него е валидно или не. С помощта на **System.out**, ако **isValidTime** е true, в конзолата извеждаме "**The time is HH:mm now.**", където **HH** е съответно стойността на променливата **hours**, а **mm** – тази на променливата **minutes**. Съответно в **else** частта от условната конструкция извеждаме, че въведеното време е невалидно – "**Incorrect time!**".

Ето как изглежда изходът от програмата при въвеждане на коректни данни:

```
What time is it?
Hours: 17
Minutes: 33
The time is 17:33 now.
```

Ето какво се случва при въвеждане на некоректни данни:

```
What time is it?
Hours: 33
Minutes: -2
Incorrect time!
```

Сортиране на числа – пример

Нека се опитаме да създадем метод, който сортира във възходящ ред подадени му числа и като резултат връща масив със сортираните числа.

При тази формулировка на задачата, се досещаме, че подзадачите, с които трябва да се справим са две:

- По какъв начин да подадем на нашия метод числата, които трябва да сортираме.
- Как да извършим сортирането на тези числа.

Това, че трябва да върнем като резултат от изпълнението на метода, масив със сортираните числа, ни подсказва, че може да декларираме метода да приема масив от числа, който масив в последствие да сортираме, а след това да върнем като резултат:

```
public static int[] sort(int[] numbers) {
   // The sorting logic comes here...

   return numbers;
}
```

Това решение изглежда, че удовлетворява изискванията от задачата ни, но се досещаме, че може да го оптимизираме малко и вместо метода да приема

като един аргумент числов масив, може да го декларираме, да приема произволен брой числови параметри.

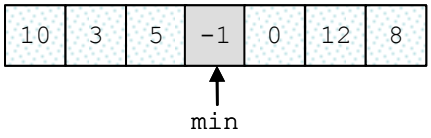
Това ще ни спести предварителното инициализиране на масив преди извикването на метода при по-малък брой числа за сортиране, а когато числата са по-голям брой, както видяхме в секцията за деклариране ма метод с произволен брой аргументи, директно можем да подадем на метода инициализиран масив от числа, вместо да ги изброяваме като параметри на метода. Така първоначалната декларация на метода ни приема следния вид:

```
public static int[] sort(int... numbers) {
   // The sorting logic comes here...

   return numbers;
}
```

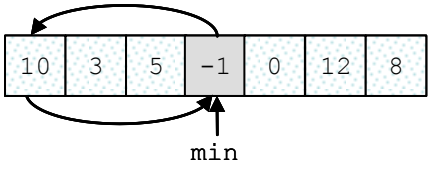
Сега трябва да решим как да сортираме нашия масив. Един от най-лесните начини това да бъде направено е чрез така наречения **метод на пряката селекция**. При него масива се разделя на сортирана и несортирана част. Сортираната част се намира в лявата част на масива, а несортираната – в дясната. При всяка стъпка на алгоритъма, сортираната част се разширява надясно с един елемент, а несортираната – намалява с един от ляво.

Нека разгледаме паралелно с обясненията един пример. Нека имаме следния несортиран масив от числа:

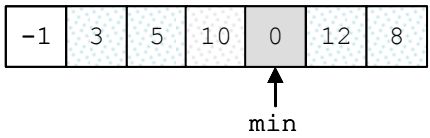
При всяка стъпка, нашият алгоритъм трябва да намери минималния елемент в несортираната част на масива:



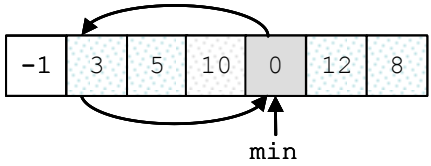
След това, трябва да размени намерения минимален елемент с първия елемент от несортираната част на масива:

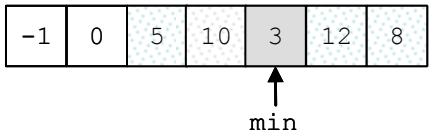


След което, отново се търси минималният елемент в оставащата несортирана част на масива:

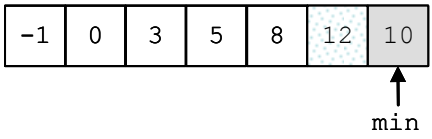


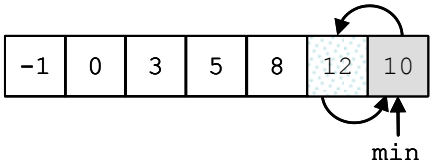
Тя се разменя с първия елемент от оставащата несортирана част:



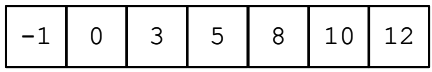


Тази стъпка се повтаря, докато несортираната част на масива не бъде изчерпана:





Накрая масивът е сортиран:



Ето какъв вид добива нашия метод, след имплементацията на току що описания алгоритъм:

```
public static int[] sort(int... numbers) {

   // The sorting logic:
   for (int i = 0; i < numbers.length - 1; i++) {

      // Loop that is operating with the un-sorted part of
```

```
      // the array
      for (int j = i + 1; j < numbers.length; j++) {

         // Swapping the values
         if (numbers[i] > numbers[j]) {
            int tempVar = numbers[i];
            numbers[i] = numbers[j];
            numbers[j] = tempVar;
         }
      }
   } // End of the sorting logic
   return numbers;
}
```

Нека декларираме и един метод **printNumbers(int...)** за извеждане на списъка с числа в конзолата:

SortingEngine.java

```
public class SortingEngine {

   public static int[] sort(int... numbers) {

      // The sorting logic:
      for (int i = 0; i < numbers.length - 1; i++) {

         // Loop that is operating over the un-sorted part of
         // the array
         for (int j = i + 1; j < numbers.length; j++) {

            // Swapping the values
            if (numbers[i] > numbers[j]) {
               int temp = numbers[i];
               numbers[i] = numbers[j];
               numbers[j] = temp;
            }
         }
      } // End of the sorting logic
      return numbers;
   }

   public static void printNumbers(int... numbers) {
      for (int i = 0; i < numbers.length; i++) {
         System.out.printf("%d", numbers[i]);
         if (i < (numbers.length - 1)) {
            System.out.print(", ");
         }
```

```
        }
    }

    public static void main(String[] args) {
        int[] numbers = sort(10, 3, 5, -1, 0, 12, 8);
        printNumbers(numbers);
    }
}
```

Съответно, след компилирането и изпълнението на този код, резултатът е точно този, който очакваме:

```
-1, 0, 3, 5, 8, 10, 12
```

Утвърдени практики при работа с методи

Въпреки че в главата "Качествен програмен код" ще обясним повече за това, нека прегледаме едни някои основни правила при работа с методи, които показват добър стил на програмиране. Ето някои от тях:

- Всеки метод трябва да решава самостоятелна, добре дефинирана задача. Това свойство се нарича **strong cohesion**. Фокусирането върху една, единствена задача позволява кодът да бъде по-лесен за разбиране и да се поддържа по-лесно. Един метод не трябва да решава няколко задачи едновременно!
- Един метод трябва да име, което описва какво прави той. Примерно метод, който сортира числа, трябва да се казва **`sortNumbers()`**, а не **`number()`** или **`processing()`** или **`method2()`**. Ако не можете да измислите подходящо име за даден метод, то най-вероятно методът решава повече от една задачи и трябва да се раздели на няколко отделни метода.
- Имената на методите е препоръчително да бъдат съставени от глагол или от глагол и съществително име (евентуално с прилагателно, което пояснява съществителното), примерно **`findSmallestElement()`** или **`sort(int[] arr)`** или **`readInputData()`**.
- Имената на методите в Java е прието да започват с малка буква. Използва се правилото **`camelCase`**, т.е. всяка нова дума, която се долепя в задната част на името на метода, започва с главна буква.
- Един метод или трябва да свърши работата, която е описана от името му, или трябва да съобщи за грешка. Не е коректно методите да връщат грешен или странен резултат при некоректни входни данни. Методът или решава задачата, за която е предназначен, или връща грешка. Всякакво друго поведение е грешно. Ще обясним в детайли по какъв начин методите могат да съобщават за грешки в главата "Обработка на изключения".

- Един метод трябва да бъде минимално обвързан с обкръжаващата го среда (най-вече с класа, в който е дефиниран). Това означава, че методът трябва да обработва данни, идващи като параметри, а не данни, достъпни по друг начин и не трябва да има странични ефекти (например да промени някоя глобално достъпна променлива). Това свойство на методите се нарича **loose coupling**.
- Трябва да се избягват методи, които са по-дълги от "един екран". За да се постигне това, логиката имплементирана в метода, се разделя по функционалност на няколко по-малки метода и след това тези методи се извикват в "дългия" до момента метод.
- Понякога, за да се подобри четимостта и прегледността на кода, е добре функционалност, която е добре обособена логически, да се отделя в метод. Например, ако имаме метод за намиране на лице на квадрат, процесът на пресмятане на квадрат на едно число може да се дефинира в отделен метод и след това, този нов метод, да се извика от метода, който пресмята лицето на фигурата квадрат. Разбира се, това ще ни даде възможност да преизползваме метода за намиране на квадрата на едно число и на други места, когато ни е нужно.

Упражнения

1. Напишете метод, който при подадено име отпечатва в конзолата "**Hello, <name>!**" (например "**Hello, Peter!**"). Напишете програма, която тества този метод.
2. Създайте метод **getMax()** с два целочислени (**int**) параметъра, който връща по-голямото от двете числа. Напишете програма, която прочита три цели числа от конзолата и отпечатва най-голямото от тях, използвайки метода **getMax()**.
3. Напишете метод, който връща английското наименование на последната цифра от дадено число. Примери: за числото **512** отпечатва "**two**"; за числото **1024** – "**four**".
4. Напишете метод, който намира колко пъти дадено число се среща в даден масив. Напишете програма, която проверява дали този метод работи правилно.
5. Напишете метод, който проверява дали елемент, намиращ се на дадена позиция от масив, е по-голям, или съответно по-малък от двата му съседа.
6. Напишете метод, който връща позицията на първия елемент на масив, който е по-голям от двата свои съседи едновременно, или **-1**, ако няма такъв елемент.
7. Напишете метод, който отпечатва цифрите на дадено десетично число в обратен ред. Например **256**, трябва да бъде отпечатано като **652**.

8. Напишете програма, която пресмята и отпечатва **n!** за всяко n в интервала **[1..100]**.
9. Напишете програма, която решава следните задачи:
   - Обръща последователността на цифрите на едно число.
   - Пресмята средното аритметично на дадена редица.
   - Решава линейното уравнение **a * x + b = 0**.

   Създайте подходящи методи за всяка една от задачите.

   Напишете програмата така, че на потребителя да му бъде изведено текстово меню, от което да избира коя задача да решава.

   Направете проверка на входните данни:
   - Десетичното число трябва да е неотрицателно.
   - Редицата не трябва да е празна.
   - Коефициентът **a** не трябва да е **0**.
10. Напишете метод, който умножава два многочлена.

Решения и упътвания

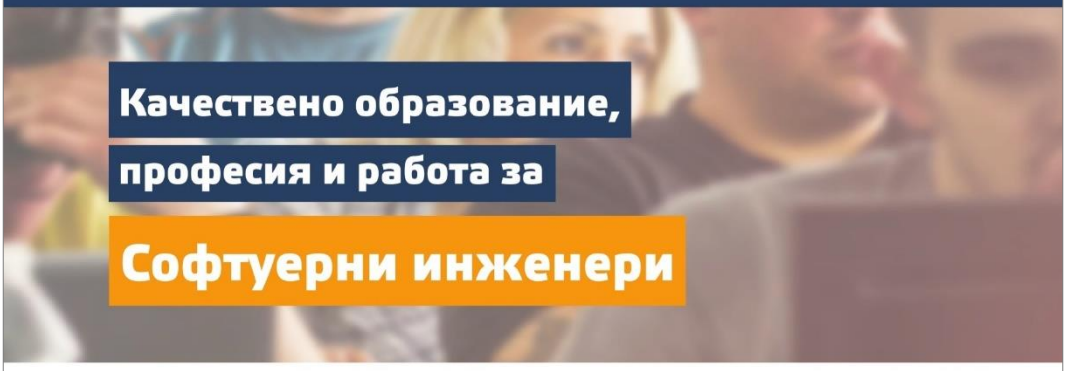
1. Използвайте метод с параметър **String**.
2. Използвайте свойството **Max(a, b, c) = Max(Max(a, b), c)**.
3. Използвайте остатъка при деление на 10 и **switch** конструкцията.
4. Методът трябва да приема като параметър масив от числа (**int[]**) и търсеното число (**int**).
5. Елементите на първа и последна позиция в масива, ще бъдат сравнявани съответно само с десния и левия си съсед.
6. Модифицирайте метода, имплементиран в предходната задача.
7. Има два начина:

   Първи начин: Нека числото е num. Докато **num ≠ 0** отпечатваме последната му цифра (**num % 10**) и след това разделяме **num** на 10.

   Втори начин: преобразуваме числото в **String** и го отпечатваме отзад напред чрез **for** цикъл.
8. Трябва да имплементирате собствен метод за умножение на големи цели числа, тъй като **100!** не може да се събере в **long**. Можете да представите числата в масив в обратен ред, с по една цифра във всеки елемент. Например числото 512 може да се представи като {2, 1, 5}. След това умножението може да го реализирате, както сте учили в училище (умножавате цифра по цифра и събирате резултатите с отместване на разрядите).

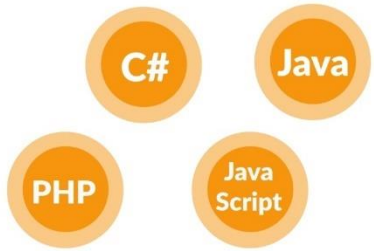
9. Създайте първо необходимите ви методи. Менюто реализирайте чрез извеждане на списък от номерирани действия (1 - обръщане, 2 - средно аритметично, 3 - уравнение) и избор на число между 1 и 3.
10. Използвайте масиви за представяне на многочлените и правилата за събиране и умножение, които познавате от математиката.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 10. Рекурсия

Автор

Радослав Иванов

Светлин Наков

В тази тема...

В настоящата тема ще се запознаем с рекурсията и нейните приложения. Рекурсията представлява мощна техника, при която един метод извиква сам себе си. С нея могат да се решават сложни комбинаторни задачи, при които с лекота могат да бъдат изчерпвани различни комбинаторни конфигурации. Ще ви покажем много примери за правилно и неправилно използване на рекурсия и ще ви убедим колко полезна може да е тя.

Какво е рекурсия?

Рекурсията е програмна техника, чиято правилна употреба води до елегантни решения на определени проблеми. Понякога нейното използване може да опрости значително кода и да подобри четимостта му.

Един обект наричаме **рекурсивен**, ако съдържа себе си или е дефиниран чрез себе си.

Рекурсия е програмна техника, при която даден **метод извиква сам себе си** при решаването на определен проблем. Такива методи наричаме **рекурсивни**.

Пример за рекурсия

Нека разгледаме числата на Фибоначи. Това са членовете на следната редица:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Всеки член на редицата се получава като сума на предходните два. Първите два члена по дефиниция са равни на 1, т.е. в сила е:

$F_1 = F_2 = 1$

$F_i = F_{i-1} + F_{i-2}$ (за $i > 2$)

Изхождайки директно от дефиницията, можем да реализираме следния рекурсивен метод за намиране на n-тото число на Фибоначи:

```
public static long fib(int n) {
   if (n <= 2)
      return 1;

   return fib(n - 1) + fib(n - 2);
}
```

Този пример ни показва, колко проста и естествена може да бъде реализацията на дадено решение с помощта на рекурсия.

От друга страна, той може да ни служи и като пример, колко трябва да сме внимателни при използването на рекурсия. Макар и интуитивно, текущото решение е един от класическите примери, когато използването на рекурсия е изключително неефективно, поради множеството изчисления (на едни и същи членове на редицата) в следствие на рекурсивните извиквания.

На предимствата и недостатъците от използване на рекурсия, ще се спрем в детайли малко по-късно в настоящата тема.

Пряка и косвена рекурсия

Когато в тялото на метод се извършва обръщение към същия метод, казваме, че методът е **пряко рекурсивен**.

Ако метод A се обръща към метод B, B към C, а C отново към A, казваме, че методът A, както и методите B и C са **непряко (косвено) рекурсивни** или **взаимно-рекурсивни**.

Веригата от извиквания при косвената рекурсия може да съдържа множество методи, както и разклонения, т.е. при наличие на едно условие се извиква един метод, а при различно условие се извиква друг.

Дъно на рекурсията

Реализирайки рекурсия, трябва да сме сигурни, че след краен брой стъпки ще получим конкретен резултат. Затова трябва да имаме поне един случай, чието решение можем да намерим директно, без рекурсивно извикване. Тези случаи наричаме **дъно на рекурсията**.

В примера с числата на Фибоначи, дъното на рекурсията е случаят, когато **n** е по-малко или равно на 2. При него можем директно да върнем резултат, без да извършваме рекурсивни извиквания, тъй като по дефиниция първите два члена на редицата на Фибоначи са равни на 1.

Ако даден рекурсивен метод няма дъно на рекурсията, тя става безкрайна и резултатът е **StackOverflowException**.

Създаване на рекурсивни методи

Когато създаваме рекурсивни методи, трябва разбием задачата, която решаваме на подзадачи, за чието решение можем да използваме същия алгоритъм (рекурсивно).

Комбинирането на решенията на всички подзадачи, трябва да води до решение на изходната задача.

При всяко рекурсивно извикване, проблемната област трябва да се ограничава така, че в даден момент да достигнем дъното на рекурсията, т.е. всички подзадачи трябва да се стремят да достигнат дъното на рекурсията.

Рекурсивно изчисляване на факториел

Използването на рекурсия ще илюстрираме с един класически пример – рекурсивно изчисляване на факториел.

Факториел от n (записва се n!) е произведението на естествените числа от 1 до n, като по дефиниция 0! = 1.

n! = 1.2.3…n

Рекурентна дефиниция

При създаването на нашето решение, много по-удобно е да използваме съответната рекурентна дефиниция на факториел:

n! = 1, при n = 0

n! = n.(n-1)! за n>0

Намиране на рекурентна зависимост

Наличието на рекурентна зависимост не винаги е очевидно. Понякога се налага сами да я открием. В нашия случай можем да направим това, анализирайки проблема и пресмятайки стойностите на факториел за първите няколко естествени числа.

```
0! = 1
1! = 1 = 1.1 = 1.0!
2! = 2.1 = 2.1!
3! = 3.2.1 = 3.2!
4! = 4.3.2.1 = 4.3!
5! = 5.4.3.2.1 = 5.4!
```

От тук лесно се вижда рекурентната зависимост:

```
n! = n.(n-1)!
```

Реализация на алгоритъма

Дъното на нашата рекурсия е простият случай n = 0, когато стойността на факториел е 1.

В останалите случай, решаваме задачата за n-1 и умножаваме получения резултат по n. Така след краен брой стъпки, със сигурност ще достигнем дъното на рекурсията, понеже между 0 и n има краен брой естествени числа.

След като имаме налице тези ключови условия, можем да реализираме метод изчисляващ факториел.

```
public static long factorial(int n) {
   // The bottom of the recursion
   if (n == 0) {
      return 1;
   }
   // Recursive call: the method calls itself
   else {
      return n * factorial(n - 1);
   }
}
```

Използвайки този метод, можем да създадем приложение, което чете от конзолата цяло число, изчислява факториела му и отпечатва получената стойност:

RecursiveFactorial.java

```
import java.util.Scanner;

public class RecursiveFactorial {

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("n = ");
      int n = input.nextInt();

      long factorial = factorial(n);
      System.out.printf("%d! = %d%n", n, factorial);
      input.close();
   }

   public static long factorial(int n) {
      // The bottom of the recursion
      if (n == 0) {
         return 1;
      }
```

```
      // Recursive call: the method calls itself
      else {
         return n * factorial(n - 1);
      }
   }
}
```

Ето какъв ще е резултатът от изпълнението на приложението, ако въведем 5 за стойност на n:

```
n = 5
5! = 120
```

Рекурсия или итерация

Изчислението на факториел често се дава като пример при обяснението на понятието рекурсия, но в този случай, както и в редица други, рекурсията далеч не е най-добрия подход.

Често, ако е зададена рекурентна дефиниция на проблема, рекурентното решение е интуитивно и не представлява трудност, докато **итеративно** (последователно) решение не винаги е очевидно.

В конкретния случай, реализацията на итеративно решение е също толкова кратка, но по-ефективна:

```
public static long factorial(int n) {
   long result = 1;

   for (int i = 1; i <= n; i++) {
      result = result * i;
   }

   return result;
}
```

Предимствата и недостатъците при използването на рекурсия и итерация ще разгледаме малко по-нататък в настоящата тема.

Сага трябва да запомним, че преди да пристъпим към реализацията на рекурсивно решение, трябва да помислим и за итеративен вариант, след което да изберем по-доброто решение според конкретната ситуация.

Нека се спрем на още един пример, където можем да използваме рекурсия за решаване на проблема, като ще разгледаме и итеративно решение.

Имитация на N вложени цикъла

Често се налага да пишем вложени цикли. Когато те са два, три или друг предварително известен брой, това става лесно. Ако броят им, обаче, не е предварително известен, се налага да търсим алтернативен подход. Такъв е случаят в следващата задача.

Да се напише програма, която симулира изпълнението на **N** вложени цикъла от **1** до **K**, където **N** и **K** се въвеждат от потребителя. Резултатът от изпълнението на програмата, трябва да е еквивалентен на изпълнението на следния фрагмент:

```
for (a1 = 1; a1 <= K; a1++)
   for (a2 = 1; a2 <= K; a2++)
      for (a3 = 1; a3 <= K; a3++)
         ...
         for (aN = 1; aN <= K; aN++)
            System.out.printf("%d %d %d ... %d \n",
                  a1, a2, a3, ... , aN);
```

Например при N = 2 и K = 3 (което е еквивалентно на 2 вложени цикъла от 1 до 3) и при N = 3 и K = 3, резултатите трябва да са съответно:

```
                1 1                     1 1 1
                1 2                     1 1 2
                1 3                     1 1 3
  N = 2     2 1       N = 3     1 2 1
  K = 3->   2 2       K = 3->   ...
                2 3                     3 2 3
                3 1                     3 3 1
                3 2                     3 3 2
                3 3                     3 3 3
```

Алгоритъмът за решаване на тази задача не е така очевиден, както в предния пример. Нека разгледаме две различни решения – едното рекурсивно, а другото итеративно.

Всеки ред от резултата, можем да разглеждаме като наредена последователност от N числа. Първото число представлява текущата стойност на брояча на първия цикъл, второто на втория и т.н. На всяка една позиция, можем да имаме стойност между 1 и K. Решението на нашата задача се свежда до намирането на всички наредени N-торки за дадени N и K.

Вложени цикли – рекурсивен вариант

Първият проблем, който се изправя пред нас е намирането на рекурентна зависимост. За тази цел, нека се вгледаме малко по-внимателно в примера от условието на задачата.

Забелязваме, че ако сме пресметнали решението за N = 2, то решението за N = 3 можем да получим, като поставим на първа позиция всяка една от стойностите на K (в случая от 1 до 3), а на останалите 2 позиции поставяме последователно всяка от двойките числа, получени от решението за N = 2. Можем да проверим, че това правило важи и при стойности на N по-големи от 3.

Така получаваме следната зависимост – започвайки от първа позиция, поставяме на текущата позиция всяка една от стойностите от 1 до K и продължаваме рекурсивно със следващата позиция. Това продължава, докато достигнем позиция N, след което отпечатваме получения резултат. Ето как изглежда и съответният метод на Java:

```
public static void nestedLoops(int currentLoop) {
   if (currentLoop == numberOfLoops) {
      printLoops();
      return;
   }

   for (int counter=1; counter<=numberOfIterations; counter++) {
      loops[currentLoop] = counter;
      nestedLoops(currentLoop + 1);
   }
}
```

Последователността от стойности ще пазим в масив наречен **loops**, който при нужда ще бъде отпечатван от метода **printLoops()**.

Методът **nestedLoops(…)** има един параметър, указващ текущата позиция, на която ще поставяме стойности.

В цикъла, поставяме последователно на текущата позиция всяка една от възможните стойности (променливата **numberOfIterations** съдържа стойността на K въведена от потребителя), след което извикваме рекурсивно метода **nestedLoops(…)** за следващата позиция.

Дъното на рекурсията се достига, когато текущата позиция стане равна на N (променливата **numberOfLoops** съдържа стойността на N въведена от потребителя). В този момент имаме стойности на всички позиции и отпечатваме последователността.

Ето и цялостна реализация на решението:

RecursiveNestedLoops.java

```
import java.util.Scanner;

public class RecursiveNestedLoops {
   public static int numberOfLoops;
```

```
   public static int numberOfIterations;
   public static int[] loops;

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      System.out.print("N = ");
      numberOfLoops = input.nextInt();

      System.out.print("K = ");
      numberOfIterations = input.nextInt();

      input.close();

      loops = new int[numberOfLoops];

      nestedLoops(0);
   }

   public static void nestedLoops(int currentLoop) {
      if (currentLoop == numberOfLoops) {
         printLoops();
         return;
      }

      for (int counter=1;counter<=numberOfIterations;counter++) {
         loops[currentLoop] = counter;
         nestedLoops(currentLoop + 1);
      }
   }

   public static void printLoops() {
      for (int i = 0; i < numberOfLoops; i++) {
         System.out.printf("%d ", loops[i]);
      }
      System.out.println();
   }
}
```

Ако стартираме приложението и въведем за стойности на **N** и **K** съответно 2 и 4, ще получим следния резултат:

```
N = 2
K = 4
1 1
1 2
1 3
```

```
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

В метода **`main(…)`** въвеждаме стойности за **`N`** и **`K`**, създаваме масива, в който ще пазим последователността от стойности, след което извикваме метода **`nestedLoops(…)`**, започвайки от първа позиция.

Забележете, че като параметър на метода подаваме 0, понеже пазим последователността от стойности в масив, а както вече знаем, номерацията на елементите в масив започва от 0.

Методът **`printLoops()`** обхожда всички елементи на масива и ги отпечатва на конзолата.

Вложени цикли – итеративен вариант

За реализацията на итеративно решение, можем да използваме следния алгоритъм, който на всяка итерация намира следващата последователност от числа и я отпечатва:

1. В начално състояние на всички позиции поставяме числото 1.
2. Отпечатваме текущата последователност от числа.
3. Увеличаваме с единица числото намиращо се на позиция N. Ако получената стойност е по-голяма от K, заменяме я с 1 и увеличаваме с единица стойността на позиция N-1. Ако и нейната стойност е станала по-голяма от K, също я заменяме с 1 и увеличаваме с единица стойността на позиция N-2 и т.н.
4. Ако стойността на първа позиция, е станала по-голяма от K, алгоритъмът приключва работа.
5. Преминаваме към стъпка 2.

Следва примерна реализация на описания алгоритъм:

`IterativeNestedLoops.java`

```
import java.util.Scanner;

public class IterativeNestedLoops {
   public static int numberOfLoops;
   public static int numberOfIterations;
   public static int[] loops;

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      System.out.print("N = ");
      numberOfLoops = input.nextInt();

      System.out.print("K = ");
      numberOfIterations = input.nextInt();

      input.close();

      loops = new int[numberOfLoops];

      nestedLoops();
   }

   public static void nestedLoops() {
      initLoops();

      int currentPosition;

      while (true) {
         printLoops();

         currentPosition = numberOfLoops - 1;
         loops[currentPosition] = loops[currentPosition] + 1;

         while (loops[currentPosition] > numberOfIterations) {
            loops[currentPosition] = 1;
            currentPosition--;

            if (currentPosition < 0) {
               return;
            }
            loops[currentPosition] = loops[currentPosition] + 1;
         }
      }
   }

   public static void initLoops() {
      for (int i = 0; i < numberOfLoops; i++) {
```

```
            loops[i] = 1;
        }
    }

    public static void printLoops() {
        for (int i = 0; i < numberOfLoops; i++) {
            System.out.printf("%d ", loops[i]);
        }
        System.out.println();
    }
}
```

Методите **main(…)** и **printLoops()** са същите, както в реализацията на рекурсивното решение.

Различен е метода **nestedLoops()**, който сега реализира алгоритъма за итеративно решаване на проблема и поради това не приема параметър, както в рекурсивния вариант.

В самото начало на този метод извикваме метода **initLoops()**, който обхожда елементите на масива и поставя на всички позиции единици.

Стъпките на алгоритъма реализираме в безкраен цикъл, от който ще излезем в подходящ момент, прекратявайки изпълнението на метода чрез оператора **return**.

Интересен е начинът, по който реализираме стъпка 3 от алгоритъма. Проверката за стойности по-големи от K, заменянето им с единица и увеличаването на стойността на предходна позиция (след което правим същата проверка и за нея), реализираме с помощта на един **while** цикъл, в който влизаме само ако стойността е по-голяма от K.

За целта първо заменяме стойността на текущата позиция с единица. След това текуща става позицията преди нея. После увеличаваме стойността на новата позиция с единица и се връщаме в началото на цикъла. Тези действия продължават, докато стойността на текуща позиция не се окаже по-малка или равна на K (променливата **numberOfIterations** съдържа стойността на K), при което излизаме от цикъла.

В момента, когато на първа позиция стойността стане по-голяма от K (това е моментът, когато трябва да приключим изпълнението), на нейно място поставяме единица и опитваме да увеличим стойността на предходната позиция. В този момент стойността на променливата **currentPosition** става отрицателна (понеже първата позиция в масив е 0), при което прекратяваме изпълнението на метода чрез оператора **return**. С това задачата ни е изпълнена.

Кога да използваме рекурсия и кога итерация?

Когато алгоритъмът за решаване на даден проблем е рекурсивен, реализирането на рекурсивно решение, може да бъде много по-четливо и елегантно от реализирането на итеративно решение на същия проблем.

Понякога дефинирането на еквивалентен итеративен алгоритъм е значително по-трудно и не е лесно да се докаже, че двата алгоритъма са еквивалентни.

В определени случаи, чрез използването на рекурсия, можем да постигнем много по-прости, кратки и лесни за разбиране решения.

От друга страна, рекурсивните извиквания, може да консумират много повече ресурси и памет. При всяко рекурсивно извикване, в стека се заделя нова памет за аргументите, локалните променливи и връщаните резултати. При прекалено много рекурсивни извиквания може да се получи препълване на стека, поради недостиг на памет.

В дадени ситуации рекурсивните решения може да са много по-трудни за разбиране и проследяване от съответните итеративни решения.

Рекурсията е мощна програмна техника, но трябва внимателно да преценяваме, преди да я използваме. При неправилна употреба, тя може да доведе до неефективни и трудни за разбиране и поддръжка решения.

Ако чрез използването на рекурсия, постигаме по-просто, кратко и по-лесно за разбиране решение, като това не е за сметка на ефективността и не предизвиква други странични ефекти, тогава можем да предпочетем рекурсивното решение. В противен случай, е добре да помислим дали не е по-подходящо да използваме итерация.

Числа на Фибоначи – защо рекурсията е неефективна?

Нека се върнем отново към примера с намирането на n-тото число на Фибоначи и да разгледаме по-подробно рекурсивното решение:

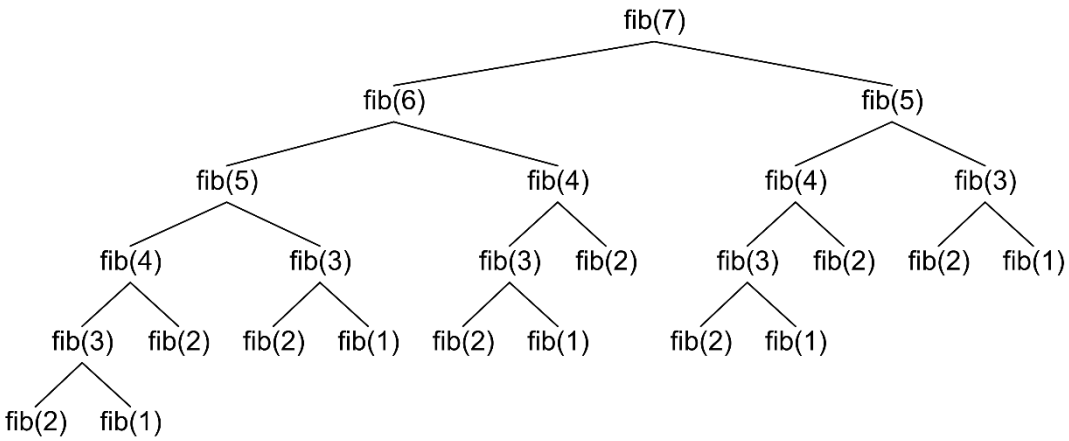
```
public static long fib(int n) {
   if (n <= 2) {
      return 1;
   }
   return fib(n - 1) + fib(n - 2);
}
```

Това решение е интуитивно, кратко и лесно за разбиране. На пръв поглед изглежда, че това е чудесен пример за приложение на рекурсията. Истината е, че това е един от класическите примери за неподходящо използване на рекурсия. Нека стартираме следното приложение:

RecursiveFibonacci.java

```
import java.util.Scanner;

public class RecursiveFibonacci {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("n = ");
      int n = input.nextInt();

      long result = fib(n);
      System.out.printf("F%d = %d%n", n, result);
      input.close();
   }

   public static long fib(int n) {
      if (n <= 2) {
         return 1;
      }
      return fib(n - 1) + fib(n - 2);
   }
}
```

Ако зададем като стойност n = 100, изчисленията ще отнемат много дълго време (едва ли някой ще изчака толкова дълго, че да види резултата). Причината за това е, че подобна реализация е изключително неефективна. Всяко рекурсивно извикване води след себе си още две, при което дървото на извикванията расте експоненциално, както е показано на фигурата:



Броят на стъпките за изчисление на fib(100) е от порядъка на 1.6 на степен 100 (това се доказва математически), докато при линейно решение е само 100.

Проблемът произлиза от това, че се правят напълно излишни изчисления. Много от членовете на редицата се пресмятат многократно.

Числа на Фибоначи – ефективна рекурсия

Можем да оптимизираме рекурсивния метод за изчисление на числата на Фибоначи, като записваме вече пресметнатите числа в масив и извършваме рекурсивно извикване само ако числото, което пресмятаме, не е било все до пресметнато до момента. Благодарение на тази малка оптимизация, рекурсивното решение ще работи с линейна сложност. Ето примерна реализация:

RecursiveFibonacciMemoization.java

```
import java.util.Scanner;

public class RecursiveFibonacciMemoization {
   public static long[] numbers;

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("n = ");
      int n = input.nextInt();

      numbers = new long[n + 2];
      numbers[1] = 1;
      numbers[2] = 1;

      long result = fib(n);
      System.out.printf("F%d = %d%n", n, result);
      input.close();
   }

   public static long fib(int n) {
      if (0 == numbers[n]) {
         numbers[n] = fib(n - 1) + fib(n - 2);
      }

      return numbers[n];
   }
}
```

Забелязвате ли разликата? Докато при първоначалния вариант, при n = 100, ни се струва, че изчисленията продължават безкрайно дълго, при оптимизираното решение, получаваме отговор мигновено.

Числа на Фибоначи – итеративно решение

Не е трудно да забележим, че можем да решим проблема и без използването на рекурсия, пресмятайки числата на Фибоначи последователно. За целта ще пазим само последните два пресметнати члена на редицата и чрез тях ще получаваме следващия. Следва реализация на итеративния алгоритъм:

IterativeFibonacci.java

```
import java.util.Scanner;

public class IterativeFibonacci {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("n = ");
      int n = input.nextInt();

      long result = fib(n);
      System.out.printf("F%d = %d%n", n, result);
      input.close();
   }

   public static long fib(int n) {
      long fn = 1;
      long fn_1 = 1;
      long fn_2 = 1;

      for (int i = 2; i < n; i++) {
         fn = fn_1 + fn_2;

         fn_2 = fn_1;
         fn_1 = fn;
      }

      return fn;
   }
}
```

Това решение е също толкова кратко и елегантно, но не крие рисковете от използването на рекурсия. Освен това то е ефективно и не изисква допълнителна памет.

Изхождайки от горните примери, можем да дадем следната препоръка:

Избягвайте рекурсията, освен, ако не сте сигурни как работи тя и какво точно се случва зад кулисите. Рекурсията е голямо и мощно оръжие, с което лесно можете да се застреляте в крака. Ползвайте я внимателно!

Ако следваме това правило, ще намалим значително вероятността за неправилно използване на рекурсия и последствията, произтичащи от него.

Още за рекурсията и итерацията

По принцип, когато имаме **линеен изчислителен процес**, не трябва да използваме рекурсия, защото итерацията може да се реализира изключително лесно и води до прости и ефективни изчисления. Пример за линеен изчислителен процес е изчислението на факториел. При него изчисляваме членовете на редица, в която всеки следващ член зависи единствено от предходните.

Линейните изчислителни процеси се характеризират с това, че на всяка стъпка от изчисленията рекурсията се извиква еднократно, само в една посока. Схематично линейният изчислителен процес можем да опишем така:

```
void recursion(parameters) {
   do some calculations;
   recursion(some parameters);
   do some calculations;
}
```

При такъв процес, когато имаме само едно рекурсивно извикване с тялото на рекурсивния метод, не е нужно да ползваме рекурсия, защото итерацията е очевидна.

Понякога, обаче имаме разклинен или **дървовиден изчислителен процес**. Например имитацията на N вложени цикъла не може лесно да се замени с итерация. Вероятно сте забелязали, че нашият итеративен алгоритъм, който имитира вложените цикли работи на абсолютно различен принцип. Опитайте да реализирате същото поведение без рекурсия и ще се убедите, че не е лесно.

По принцип всяка рекурсия може да се сведе до итерация чрез използване на стек на извикванията (какъвто се създава по време на изпълнение на програмата), но това е сложно и от него няма никаква полза. Рекурсията трябва да се ползва, когато дава просто, лесно за разбиране и ефективно решение на даден проблем, за който няма очевидно итеративно решение.

При дървовидните изчислителни процеси на всяка стъпка от рекурсията, се извършват няколко на брой рекурсивни извиквания и схемата на извършване на изчисленията може да се визуализира като дърво (а не като списък, както при линейните изчисления). Например при изчислението на числата на Фибоначи видяхме какво дърво на рекурсивните извиквания се получава.

Типичната схема на дървовидния изчислителен процес можем да опишем чрез псевдокод така:

```
void recursion(parameters) {
   do some calculations;
   recursion(some parameters);
   ...
   recursion(some other parameters);
   do some calculations;
}
```

Дървовидните изчислителни процеси не могат директно да бъдат сведени до рекурсивни (за разлика от линейните). Случаят с числата на Фибоначи е простичък, защото всяко следващо число се изчислява чрез предходните, които можем да изчислим предварително. Понякога, обаче всяко следващо число се изчислява не само чрез предходните, а и чрез следващите и рекурсивната зависимост не е толкова проста. В такъв случай рекурсията се оказва особено ефективна.

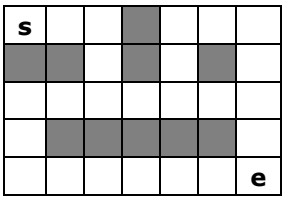
Ще илюстрираме последното твърдение с един класически пример.

Търсене на пътища в лабиринт – пример

Даден е лабиринт, който има правоъгълна форма и се състои от N*M квадратчета. Всяко квадратче е или проходимо, или не е проходимо. Търсач на приключения влиза в лабиринта от горния му ляв ъгъл (там е входът) и трябва да стигне до долния десен ъгъл на лабиринта (там е изходът). Търсачът на приключения може на всеки ход да се премести с една позиция нагоре, надолу, наляво или надясно, като няма право да излиза извън границите на от лабиринта и няма право да стъпва върху непроходими квадратчета. Преминаването през една и съща позиция повече от веднъж също е забранено (счита се, че търсачът на приключения се е загубил, ако се върне след няколко хода на място, където вече е бил). Да се напише компютърна програма, която отпечатва всички възможни пътища от началото до края на лабиринта.

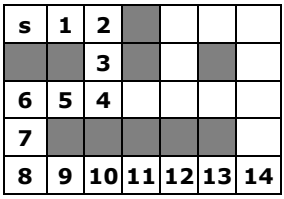
Това е типичен пример за задача, която може лесно да се реши с рекурсия, докато с итерация решението е по-сложно и по-трудно за реализация.

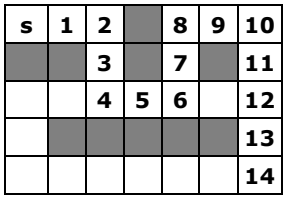
Нека първо си нарисуваме един пример, за да си представим условието на задачата и да помислим за решение:

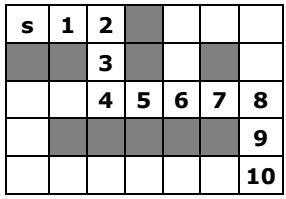


Видно е, че има 3 различни пътя от началната позиция до крайната, които отговарят на изискванията на задачата (движение само по празни

квадратчета и без преминаване по два пъти през никое от тях). Ето как изглеждат въпросните 3 пътя:







На фигурата по-горе с числата от 1 до 14 е означен номерът на съответната стъпка от пътя.

Пътища в лабиринт – рекурсивен алгоритъм

Как да решим задачата? Можем да разгледаме търсенето на дадена позиция в лабиринта до края на лабиринта като рекурсивен процес по следния начин:

- Нека текущата позиция в лабиринта е (row, col). В началото тръгваме от стартовата позиция (0,0).
- Ако текущата позиция е търсената позиция (N-1, M-1), то сме намерили път и трябва да го отпечатаме.
- Ако текущата позиция е непроходима, връщаме се назад (нямаме право да стъпваме в нея).
- Ако текущата позиция е вече посетена, връщаме се назад (нямаме право да стъпваме втори път в нея).
- В противен случай търсим път в четирите възможни посоки. Търсим рекурсивно (със същия алгоритъм) път към изхода на лабиринта като опитваме да ходим във всички възможни посоки:
    - Опитваме наляво: позиция (row, col-1).
    - Опитваме нагоре: позиция (row-1, col).
    - Опитваме надясно: позиция (row, col+1).
    - Опитваме надолу: позиция (row+1, col).

За да стигнем до този алгоритъм, разсъждаваме рекурсивно. Имаме задачата "търсене на път от дадена позиция до изхода". Тя може да се сведе до 4 подзадачи:

- търсене на път от позицията вляво от текущата до изхода;
- търсене на път от позицията нагоре от текущата до изхода;
- търсене на път от позицията вдясно от текущата до изхода;
- търсене на път от позицията надолу от текущата до изхода.

Ако от всяка възможно позиция, до която достигнем, проверим четирите възможни посоки и не се въртим в кръг (избягваме преминаване през позиция, на която вече сме били), би трябвало рано или късно да намерим изхода (ако съществува път към него).

Този път рекурсията не е толкова проста, както при предните задачи. На всяка стъпка трябва да проверим дали не сме стигнали изхода и дали не стъпваме в забранена позиция, след това трябва да отбележим позицията като посетена и да извикаме рекурсивното търсене на път в четирите посоки. След връщане от рекурсивните извиквания, трябва да отбележим обратно като непосетена позицията, от която се оттегляме. Такова обхождане е известно в информатиката като **търсене с връщане назад (backtracking)**.

Пътища в лабиринт – имплементация

За реализацията на алгоритъма ще ни е необходимо представяне на лабиринта. Ще ползваме двумерен масив от символи, като в него всяко ще означим със символа ' ' (интервал) проходимите позиции, с '**e**' изхода от лабиринта и с '*****' непроходимите полета. Стартовата позиция ще означим като празна. Позициите, през които сме минали, ще означим със символа '**s**'. Ето как ще изглежда дефиницията на лабиринта за нашия пример:

```
private static char[][] lab = {
   {' ', ' ', ' ', '*', ' ', ' ', ' '},
   {'*', '*', ' ', '*', ' ', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', ' '},
   {' ', '*', '*', '*', '*', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};
```

Нека се опитаме да реализираме рекурсивния метод за търсене в лабиринт. Той трябва да бъде нещо такова:

```
private static char[][] lab = {
   {' ', ' ', ' ', '*', ' ', ' ', ' '},
   {'*', '*', ' ', '*', ' ', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', ' '},
   {' ', '*', '*', '*', '*', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

private static void findPath(int row, int col) {
   if ((col<0) || (row<0) ||
       (col>=lab[0].length) || (row >= lab.length)) {
      // We are out of the labyrinth
      return;
   }
```

```
    // Check if we have found the exit
    if (lab[row][col] == 'e') {
        System.out.println("Found the exit!");
    }

    if (lab[row][col] != ' ') {
        // The current cell is not free
        return;
    }

    // Mark the current cell as visited
    lab[row][col] = 's';

    // Invoke recursion the explore all possible directions
    findPath(row, col-1); // left
    findPath(row-1, col); // up
    findPath(row, col+1); // right
    findPath(row+1, col); // down

    // Mark back the current cell as free
    lab[row][col] = ' ';
}

public static void main(String[] args) {
    findPath(0, 0);
}
```

Имплементацията стриктно следва описанието, дадено по-горе. В случая размерът на лабиринта не е записан в променливи N и M, а е се извлича от двумерния масив, съхраняващ лабиринта: броят колони е **lab[0]. length**, а броят редове е **lab.length** (помислете защо!).

При влизане в рекурсивния метод за търсене първо се проверява дали няма излизане извън лабиринта. Ако има търсенето от текущата позиция нататък се прекратява, защото е забранено излизане извън границите на лабиринта.

След това се проверява дали не сме намерили изхода. Ако сме го намерили, се отпечатва подходящо съобщение и търсенето от текущата позиция нататък приключва.

След това се проверява дали е свободна текущата клетка. Клетката е свободна, ако е проходима и не сме били на нея при някоя от предните стъпки (ако не е част от текущия път от стартовата позиция до текущата клетка на лабиринта).

При свободна клетка, стъпваме в нея. Означаваме клетката като заета (със символа '**s**'). След това рекурсивно търсим път в четирите възможни посоки.

След като се върнем от рекурсивното проучване на четирите възможни посоки, отстъпваме назад от текущата клетка и я маркираме като отново като свободна (връщаме се назад).

Маркирането на текущата клетка като свободна при излизане от рекурсията е важно, защото при връщане назад тя вече не е част от текущия път. Ако пропуснем това действие, няма да намерим всички пътища до изхода, а само някои от тях.

Така изглежда рекурсивният метод за търсене на изхода в лабиринта. Остава само да го извикаме от **main()** метода, започвайки търсенето на пътя от началната позиция (0, 0).

Ако стартираме програмата, ще видим следния изход:

```
Found the exit!
Found the exit!
Found the exit!
```

Вижда се, че изходът е бил намерен точно 3 пъти. Изглежда алгоритъмът работи коректно. Липсва ни обаче отпечатването на самия път като последователност от позиции, през които преминаваме.

Пътища в лабиринт – запазване на пътищата

За да можем да отпечатаме пътищата, които намираме с нашия рекурсивен алгоритъм, можем да използваме масив, в който при всяко придвижване да пазим посоката, която сме поели (**L** – наляво, **U** – нагоре, **R** – надясно, **D** – надолу). Този масив ще съдържа във всеки един момент текущия път от началото на лабиринта до текущата позиция.

Ще ни трябва един масив от символи и един брояч на стъпките, които сме направили. Броячът ще пази колко пъти сме се придвижили към следваща позиция рекурсивно, т.е. текущата дълбочина на рекурсията.

За да работи всичко коректно, е необходимо преди влизане в рекурсия да увеличаваме брояча и да запазваме посоката, която сме поели в текущата позиция от масива, а при връщане от рекурсията да намаляваме брояча. При намиране на изхода можем да отпечатаме пътя (всички символи от масива от 0 до позицията, която броячът сочи.

Колко голяма да бъде масивът? Отговорът на този въпрос е лесен; понеже в една клетка можем да влезем най-много веднъж, то никога пътят няма да е по-дълъг от общия брой клетки в лабиринта (N*M). В нашия случай размерът е 7*5, т.е. масивът е достатъчно да има 35 позиции.

Следва една примерна имплементация на описаната идея:

```
private static char[][] lab = {
  {' ', ' ', ' ', '*', ' ', ' ', ' '},
```

```
      {'*', '*', ' ', '*', ' ', '*', ' '},
      {' ', ' ', ' ', ' ', ' ', ' ', ' '},
      {' ', '*', '*', '*', '*', '*', ' '},
      {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
   };

   private static char[] path =
      new char[lab[0].length * lab.length];
   private static int position = 0;

   private static void findPath(int row, int col, char direction) {
      if ((col<0) || (row<0) ||
          (col>=lab[0].length) || (row >= lab.length)) {
         // We are out of the labyrinth
         return;
      }

      // Append the direction to the path
      path[position] = direction;
      position++;

      // Check if we have found the exit
      if (lab[row][col] == 'e') {
         printPath(path, 1, position-1);
      }

      if (lab[row][col] == ' ') {
         // The current cell is free. Mark it as visited
         lab[row][col] = 's';

         // Invoke recursion the explore all possible directions
         findPath(row, col-1, 'L'); // left
         findPath(row-1, col, 'U'); // up
         findPath(row, col+1, 'R'); // right
         findPath(row+1, col, 'D'); // down

         // Mark back the current cell as free
         lab[row][col] = ' ';
      }

      // Remove the direction from the path
      position--;
   }

   private static void printPath(
         char[] path, int startPos, int endPos) {
      System.out.print("Found path to the exit: ");
      for (int pos = startPos; pos<=endPos; pos++) {
```

```
      System.out.print(path[pos]);
   }
   System.out.println();
}

public static void main(String[] args) {
   findPath(0, 0, 'S');
}
```

За леснота добавихме още един параметър на рекурсивния метод за търсене на път до изхода от лабиринта: посока, в която сме поели, за да дойдем на текущата позиция. Този параметър няма смисъл при първоначалното започване от стартовата позиция и затова в началото слагаме за посока някаква безсмислена стойност '**S**'. След това при отпечатването пропускаме първия елемент от пътя.

Ако стартираме програмата, ще получим трите възможни пътя от началото до края на лабиринта:

```
Found path to the exit: RRDDLLDDRRRRRR
Found path to the exit: RRDDRRUURRDDDD
Found path to the exit: RRDDRRRRDD
```

Пътища в лабиринт – тестване на програмата

Изглежда алгоритъмът работи. Остава да го тестваме още малко примери, за да се убедим, че не сме допуснали някоя глупава грешка. Може да пробваме примерно с празен лабиринт с размер 1 на 1, с празен лабиринт с размер 3 на 3 и примерно с лабиринт, в който не съществува път до изхода, и накрая с огромен лабиринт, където пътищата са наистина много.

Ако изпълним тестовете, ще се убедим, че във всеки от тези необичайни случаи програмата работи коректно.

Примерен вход (лабиринт 1 на 1):

```
private static char[][] lab = {
   {'e'},
};
```

Примерен изход:

```
Found path to the exit:
```

Вижда се, че изходът е коректен, но пътят е с дължина 0, тъй като стартовата позиция съвпада с изхода. Бихме могли да визуализацията в този случай (примерно да отпечатваме "**empty path**").

Примерен вход (празен лабиринт 3 на 3):

```
private static char[][] lab = {
   {' ', ' ', ' '},
   {' ', ' ', ' '},
   {' ', ' ', 'e'},
};
```

Примерен изход:

```
Found path to the exit: RRDLLDRR
Found path to the exit: RRDLDR
Found path to the exit: RRDD
Found path to the exit: RDLDRR
Found path to the exit: RDRD
Found path to the exit: RDDR
Found path to the exit: DRURDD
Found path to the exit: DRRD
Found path to the exit: DRDR
Found path to the exit: DDRUURDD
Found path to the exit: DDRURD
Found path to the exit: DDRR
```

Вижда се, че изходът е коректен – това са всички пътища до изхода.

Примерен вход (лабиринт 5 на 3 без път до изхода):

```
private static char[][] lab = {
   {' ', '*', '*', ' ', ' '},
   {' ', ' ', ' ', '*', ' '},
   {'*', ' ', ' ', '*', 'e'},
};
```

Примерен изход:

```
(няма изход)
```

Вижда се, че изходът е коректен, но отново бихме могли да добавим по-приятелско съобщения (примерно "**No exit!**") вместо липса на какъвто и да е изход.

Сега остана да проверим какво се случва, когато имаме голям лабиринт. Ето примерен вход (лабиринт с размер 15 на 9):

```
private static char[][] lab = {
   {' ','*',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ','*','*',' ',' ',' '},
   {' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
```

```
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ','*','*','*',' ',' ','*',' ',' ',' ',' ',' ',' ',' ','*','*','*','*'},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ','e'},
};
```

Стартираме програмата и тя започва да печата непрекъснато пътища до изхода, но не свършва, защото пътищата са прекалено много. Ето как изглежда една малка част от изхода:

```
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRURDRR
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURRD
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURDR
...
```

Сега, нека пробваме един последен пример – лабиринт с голям размер (15 на 9, в който не съществува път до изхода:

```
private static char[][] lab = {
   {' ','*',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ','*','*',' ',' ',' '},
   {' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ','*','*','*',' ',' ','*',' ',' ',' ',' ',' ',' ','*','*','*','*','*'},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ','*','*',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ','*','e'},
};
```

Стартираме програмата и тя заспива, без да отпечата нищо. Всъщност работи прекалено дълго, за да я изчакаме. Изглежда имаме проблем.

Какъв е проблемът? Проблемът е, че възможните пътища, които алгоритъмът анализира, са прекалено много и това отнема прекалено много време. Да помислим колко се тези пътища. Ако средно един път до изхода е 20 стъпки и ако на всяка стъпка имаме 4 възможни посоки за продължение, то би трябвало да анализираме 4^{20} възможни пътя, което е ужасно голямо число. Тази оценка на броя възможности е изключително неточна, но дава ориентация за какъв порядък възможности става дума.

Какъв е изводът? Изводът е, че методът "**търсене с връщане назад**" не работи, когато вариантите са прекалено много, а фактът, че са прекалено много лесно може да се установи.

Няма да ви мъчим с опити да измислите решение на задачата. Проблемът за намиране на всички пътища в лабиринт няма ефективно решение при големи лабиринти.

Задачата има решение, ако бъде формулирана по друг начин: да се намери поне един изход от лабиринта. Тази задача е далеч по-лесна и може да се реши с една много малка промяна в примерния код: при връщане от рекурсията текущата позиция да не се маркира обратно като свободна. Това означава да изтрием следните редове код:

```
// Mark back the current cell as free
lab[row][col] = ' ';
```

Можем да се убедим, че след тази промяна, програмата много бързо установява, ако в лабиринта няма път до изхода, а ако има много бързо намира един от пътищата (произволен).

Използване на рекурсия – изводи

Какъв е генералният извод от задачата за търсене на път в лабиринт? Изводът вече го формулирахме: ако не разбирате как работи рекурсията, избягвайте да я ползвате! Внимавайте, когато пишете рекурсивен код. Рекурсията е много мощен метод за решаване на комбинаторни задачи (задачи, в които изчерпваме варианти), но не е за всеки. Можете много лесно да сгрешите. Лесно можете да накарате програмата да "зависне" или да препълните стека с бездънна рекурсия. Винаги търсете итеративните решения, освен, ако не разбирате в голяма дълбочина как да ползвате рекурсията!

Колкото до задачата за търсене на най-къс път в лабиринт, можете да я решите елегантно без рекурсия с т.нар. **метод на вълната**, известен още като BFS (breadth-first search), който се реализира елементарно с една опашка. Повече за алгоритъма "BFS" можете да прочетете на неговата страница в Уикипедия: http://en.wikipedia.org/wiki/Breadth-first_search.

Упражнения

1. Напишете програма, която генерира и отпечатва всички комбинации с повторение на **k** елемента над **n**-елементно множество.

   Примерен вход:

   ```
   n = 3
   k = 2
   ```

Примерен изход:

```
(1 1), (1 2), (1 3), (2 2), (2 3), (3 3)
```

2. Напишете програма, която генерира всички вариации с повторение на **n** елемента от **k**-ти клас.

   Примерен вход:

   ```
   n = 3
   к = 2
   ```

   Примерен изход:

   ```
   (1 1), (1 2), (1 3), (2 1), (2 2), (2 3), (3 1), (3 2), (3 3)
   ```

3. Нека е дадено множество от символни низове. Да се напише програма, която генерира всички подмножества съставено от **k** на брой символни низа, избрани измежду елементите на това множество.

   Примерен вход:

   ```
   strings = {'test', 'rock', 'fun'}
   k = 2
   ```

   Примерен изход:

   ```
   (test rock), (test fun), (rock fun)
   ```

4. Напишете програма, която отпечатва всички подмножества на дадено множество от думи.

   Примерен вход:

   ```
   words = {'test', 'rock', 'fun'}
   ```

   Примерен изход:

   ```
   (), (test), (rock), (fun), (test rock), (test fun),
   (rock fun), (test rock fun)
   ```

5. Реализирайте алгоритъма "сортиране чрез сливане" (merge-sort). При него началният масив се разделя на две равни по големина части, които се сортират (чрез merge-sort) и след това двете сортирани части се сливат, за да се получи целият масив в сортиран вид.

6. Напишете програма, която генерира и отпечатва пермутациите на числата 1, 2, …, n, за дадено цяло число n.

   Примерен вход:

```
n = 3
```

Примерен изход:

```
(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

7. Даден е масив с цели числа и число N. Напишете програма, която намира всички подмножества от числа от масива, които имат сума N.
8. Даден е масив с цели числа. Напишете програма, която проверява дали в масива съществуват едно или повече числа, чиято сума е N.
9. Реализирайте BFS алгоритъма за търсене на най-кратък път в лабиринт.

Решения и упътвания

1. Използвайте имитация на вложени цикли.
2. Използвайте имитация на вложени цикли.
3. Нека низовете са N на брой. Използвайте имитация на вложени цикли. Трябва да генерирате всички множества от **k** елемента в диапазона **[0...N-1]**. За всяко такова множество разглеждате числата от него като индекси в масива със символните низове и отпечатвате за всяко число съответния низ. За горния пример **{0, 2}** означава нулевата и втората дума, т.е. **(test, fun)**.
4. Можете да използвате предходната задача и да генерирате празното множество, следвано от всички подмножества с 1 елемент, всички подмножества с 2 елемента, всички множества с 3 елемента и т.н.

   Задачата има и по-хитро решение: завъртате цикъл от 0 до 2^N-1 и преобразувате всяко от тези числа в двоична бройна система. За N=3 имате следните двоични представяния на числата 0 до 2^N-1:

   ```
   000, 001, 010, 011, 100, 101, 110, 111
   ```

   Сега за всяко двоично представяне взимате тези думи от множеството символни низове, за които имаме единица на съответната позиция в двоичното представяне. Примерно за двоичното представяне "101" взимаме първия и последния низ (там имаме единици) и пропускаме втория низ (там имаме нула). Хитро, нали?
5. Ако се затрудните, потърсете "merge sort" в Интернет. Ще намерите стотици имплементации.
6. Да предположим, че методът **perm(k)** пермутира по всички възможни начини елементите от масив **p[]** на позиции от 0 до k включително. В масива **p** първоначално записваме числата от 1 до N. Можем да реализираме рекурсивно **perm(k+1)** по следния начин:

1. При **k**=0 отпечатваме предната пермутация и излизаме (дъно на рекурсията).
2. За всяка позиция **i** от 0 до k извършваме следното:
   a. Разменяме **p[i]** с **p[k]**.
   b. Извикваме рекурсия: **perm(k-1)**.
   c. Разменяме обратно **p[i]** с **p[k]**.
3. Извикваме **perm(k)**.

В началото започваме с **perm(N-1)**.

7. Задачата не се различава съществено от задачата за намиране на всички подмножества измежду даден списък със символни низове. Помислете ще работи ли бързо програмата при 500 числа?

8. Ако подходите към проблема по метода на изчерпването на всички възможности, решението няма да работи при повече от 20-30 елемента. Затова може да подходите по съвсем различен начин.

   Нека имаме масива с числа **p[]**. Нека означим с **possible(k, sum)** дали можем да получим сума **sum** като използваме само числата **p[0]**, **p[1]**, ..., **p[k]**. Тогава са в сила следните рекурентни зависимости:

   - **possible(0, sum) = true**, точно когато **p[0] == sum**
   - **possible(k, sum) = true**, точно когато **possible[k-1, sum] == true** или **possible[k-1, sum-p[k]] == true**

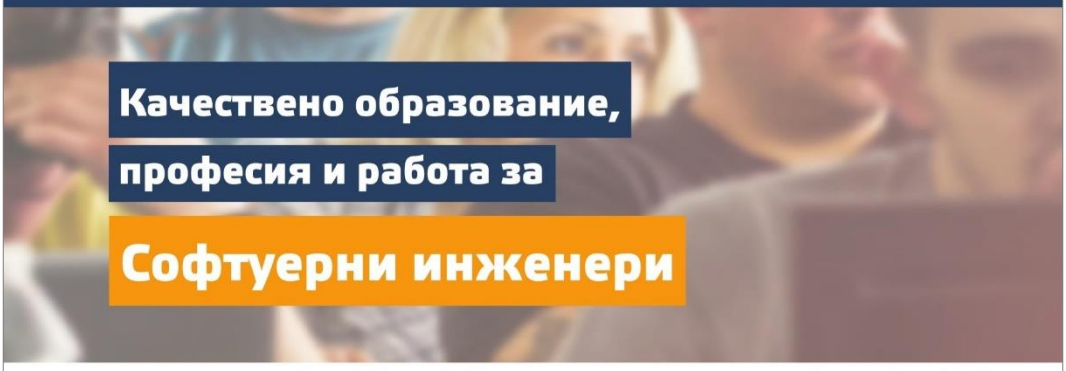
   Горната формула показва, че можем да получим сума **sum** от елементите на масива на позиции от **0** до **k**, ако едно от двете е в сила:

   - Елементът **p[k]** не участва в сумата **sum** и тя се получава по някакъв начин от останалите елементи (от **0** до **k-1**);
   - Елементът **p[k]** участва в сумата **sum**, а остатъкът **sum-p[k]** се получава по някакъв начин от останалите елементи (от **0** до **k-1**).

   Реализацията не е сложна, но не трябва да внимавате и да не позволявате вече сметната стойност от двумерния масив **possibe[][]** да се смята повторно. Иначе алгоритъмът няма да работи при над 20-30 елемента.

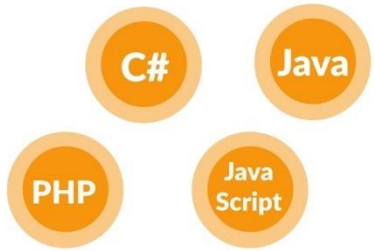
9. Прочетете статията в Уикипедия: http://en.wikipedia.org/wiki/Breadth-first_search. Там има достатъчно обяснения за BFS и примерен код.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 11. Създаване и използване на обекти

Автор

Теодор Стоев

Светлин Наков

В тази тема...

В настоящата тема ще се запознаем накратко с основните понятия в обектно-ориентираното програмиране – класовете и обектите – и ще обясним как да използваме класовете от стандартните библиотеки на Java. Ще се спрем на някои често използвани системни класове и ще видим как се създават и използват техни инстанции (обекти). Ще разгледаме как можем да осъществяваме достъп до полетата на даден обект, как да извикваме конструктори и как да работим със статичните полета в класовете. Накрая ще се запознаем с понятието пакети – какво ни помагат, как да ги включваме и използваме.

Класове и обекти

През последните няколко десетилетия програмирането и информатиката като цяло претърпяват невероятно развитие и се появяват концепции, които променят изцяло начина, по който се изграждат програми. Точно такава радикална идея въвежда **обектно-ориентираното програмиране (ООП)**. Ще изложим кратко въведение в принципите на ООП и понятията, които се използват в него. Като начало ще обясним какво представляват класовете и обектите. Тези две понятия стоят в основата на ООП и са неразделна част от ежедневието на почти всеки съвременен програмист.

Какво е обектно-ориентирано програмиране?

Обектно-ориентираното програмиране е модел на програмиране, който използва **обекти** и техните взаимодействия за изграждането на компютърни програми. По този начин се постига лесен за разбиране опростен модел на предметната област, който дава възможност на програмиста интуитивно (чрез проста логика) да решава много от задачите, които възникват в реалния свят.

Засега няма да навлизаме в детайли за това какви са целите и предимствата на ООП, както и да обясняваме подробно принципите при изграждане на йерархии от обекти. Ще вмъкнем само, че програмните техники на ООП често включват капсулация, модулност, полиморфизъм и наследяване. Тези техники са извън целите на настоящата тема, затова ще ги разгледаме по-късно в главата "Принципи на обектно-ориентираното програмиране". Сега ще се спрем на обектите като основно понятие в ООП.

Какво е обект?

Ще въведем понятието **обект** в контекста на ООП. Софтуерните обекти моделират обекти от реалния свят или абстрактни концепции (които също разглеждаме като обекти).

Примери за реални обекти са хора, коли, стоки, покупки и т.н. Абстрактните обекти са понятия в някоя предметна област, които се налага да моделираме и използваме в компютърна програма. Примери за абстрактни обекти са структурите от данни стек, опашка, списък и дърво. Те не са предмет на настоящата тема, но ще ги разгледаме в детайли по-нататък.

В обектите от реалния свят (също и в абстрактните обекти) могат да се отделят следните две групи характеристики:

- Състояния (states) – това са характеристики на обекта, които по някакъв начин го определят и описват по принцип или в конкретен момент.
- Поведения (behaviors) – това са специфични характерни действия, които обектът може да извършва.

Нека за пример вземем обектът от реалния свят "куче". Състояния на кучето могат да бъдат "име", "цвят на козината" и "порода", а негови поведения – "лаене", "седене" и "ходене".

Обектите в ООП обединяват данни и средствата за тяхната обработка в едно цяло. Те съответстват на обектите от реалния свят и съдържат в себе си данни и действия:

- Член-данни (data members) – представляват променливи, вградени в обектите, които описват състоянията им.
- Методи (methods) – вече сме ги разглеждали в детайли. Те са инструментът за изграждане на поведението на обектите.

Какво е клас?

Класът дефинира абстрактните характеристики на даден обект. Може още да се каже, че класът е план или шаблон, който описва природата на нещо (някакъв обект). Класовете са градивните елементи на ООП и са неразделно свързани с обектите. Нещо повече, всеки обект е представител на единствен точно определен клас.

Ще дадем пример за клас и обект, който е негов представител. Нека имаме клас **Dog** и обект **Lassie**, който е представител на класа **Dog** (казваме още обект от тип **Dog).** Класът **Dog** описва характеристиките на всички кучета, докато **Lassie** е конкретно куче**.**

Класовете предоставят модулност и структурност на обектно-ориентираните програми. Техните характеристики трябва да са смислени в общ контекст, така че да могат да бъдат разбрани и от хора, които са запознати с проблемната област, без да са програмисти. Например, не може класът **Dog** да има характеристика "RAM памет" поради простата причина, че в контекста на този клас такава характеристика няма смисъл.

Класове, атрибути и поведение

Класът дефинира характеристиките на даден обект (които ще наричаме **атрибути**) и неговото **поведение** (действията, които обектът може да извършва). Атрибутите на класа се дефинират като собствени променливи в тялото му (наречени член-променливи). Поведението на обектите се моделира чрез дефиниция на методи в класовете.

Ще илюстрираме казаното дотук като дадем пример за реална дефиниция на клас. Нека се върнем отново на примера с кучето, който вече дадохме по-горе. Искаме да дефинираме клас **Dog**, който моделира реалният обект "куче". Класът ще включва характеристики, общи за всички кучета (като порода и цвят на козината), а също и характерно за кучетата поведение (като лаене, седене, ходене). В такъв случай ще имаме атрибути **breed** и **furColor**, а поведението ще бъде имплементирано чрез методите **bark()**, **sit()**, **walk()**.

Обектите – инстанции на класовете

От казаното дотук знаем, че всеки обект е представител на точно един клас и е създаден по шаблон на този клас. Създаването на обект от дефиниран клас наричаме **инстанциране** (instantiation). **Инстанция** (instance) е фактическият обект, който се създава от класа по време на изпълнение на програмата.

Всеки обект е **инстанция** на конкретен клас. Тази инстанция се характеризира със **състояние** (state) – множество от стойности, асоциирани с атрибутите на класа.

В контекста на така въведените понятия, обектът се състои от две неща: моментното състояние и поведението, дефинирано в класа на обекта. Състоянието е специфично за инстанцията (обекта), но поведението е общо за всички обекти, които са представители на този клас.

Класове в Java

Дотук разгледахме някои общи характеристики на ООП. Голяма част от съвременните езици за програмиране са обектно-ориентирани. Всеки от тях има известни особености при работата с класовете и обектите. В тази книга ще се спрем само на един от тези езици – Java. Държим да отбележим, че знанията за ООП в Java ще бъдат от полза на читателя без значение кой обектно-ориентиран език използва в практиката.

Какво представляват класовете в Java?

Класът в Java се дефинира чрез ключовата дума **class**, последвана от идентификатор (име) на класа и съвкупност от член-данни и методи, обособени в собствен блок код.

Класовете в Java могат да съдържат следните елементи:

- Полета (fields) – член-променливи от определен тип;
- Свойства (properties) – това са специален вид елементи, които разширяват функционалността на полетата като дават възможност за допълнителна обработка на данните при извличането и записването им. Ще се спрем по-подробно на тях в темата "Дефиниране на класове";
- Методи – реализират манипулацията на данните.

Примерен клас

Ще дадем пример за прост клас в Java, който съдържа изброените елементи. Класът **Cat** моделира реалния обект котка и притежава свойствата име и цвят. Посоченият клас дефинира няколко полета, свойства и методи, които по-късно ще използваме наготово. Следва дефиницията на класа (засега няма да разглеждаме в детайли дефиницията на класовете – ще обърнем специално внимание на това в главата "Дефиниране на класове"):

```
public class Cat {
   // Field name
   private String name;
   // Field color
   private String color;

   // Getter of property name
   public String getName() {
      return this.name;
   }

   // Setter of property name
   public void setName(String name) {
```

```
      this.name = name;
   }

   // Getter of property color
   public String getColor() {
      return this.color;
   }

   // Setter of property color
   public void setColor(String color) {
      this.color = color;
   }

   // Default constructor
   public Cat() {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(String name, String color) {
      this.name = name;
      this.color = color;
   }

   // Method sayMiau
   public void sayMiau() {
      System.out.printf("Cat %s said: Miauuuuuu!%n", name);
   }
}
```

Извикването на метода **System.out.printf(…)** на класа **java.lang.System** е пример за употребата на **системен клас** в Java. Системни наричаме класовете, дефинирани в стандартните библиотеки за изграждане на приложения с Java (или друг език за програмиране). Те могат да се използват във всички наши приложения на Java. Такива са например класовете **String**, **System** и **Math**, които ще разгледаме малко по-късно.

Важно е да се знае, че имплементацията на класовете е **капсулирана** (скрита). При използването на методите на даден клас от приложния програмист, тяхната имплементация е независима от употребата им. При системни класове имплементация обикновено дори не е достъпна за програмиста, който ги използва. Това е така, защото за програмиста е от значение какво правят методите, а не как го правят. По този начин се създават **нива на абстракция**, което е един от основните принципи в ООП.

Ще обърнем специално внимание на системните класове малко по-късно. Сега е време да се запознаем със създаването и използването на обекти в програмите.

Създаване и използване на обекти

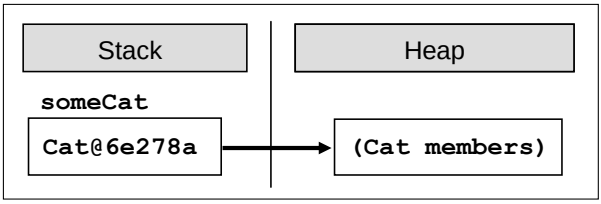
Засега ще се фокусираме върху създаването и използването на обекти в нашите програми. Ще работим с вече дефинирани или системни класове – особеностите при дефинирането на наши собствени класове ще разгледаме по-късно в темата "Дефиниране на класове".

Създаване и освобождаване на обекти

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъзданият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта. Ще отбележим, че при това присвояване същинският обект не се копира. В променливата се записва само **референция** към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

```
Cat someCat = new Cat();
```

На променливата **someCat** от тип **Cat** присвояваме новосъздадена инстанция на класа **Cat**. Променливата someCat стои в стека, а нейната стойност (инстанцията на класа **Cat**) стои в динамичната памет:



Създаване на обекти със задаване на параметри

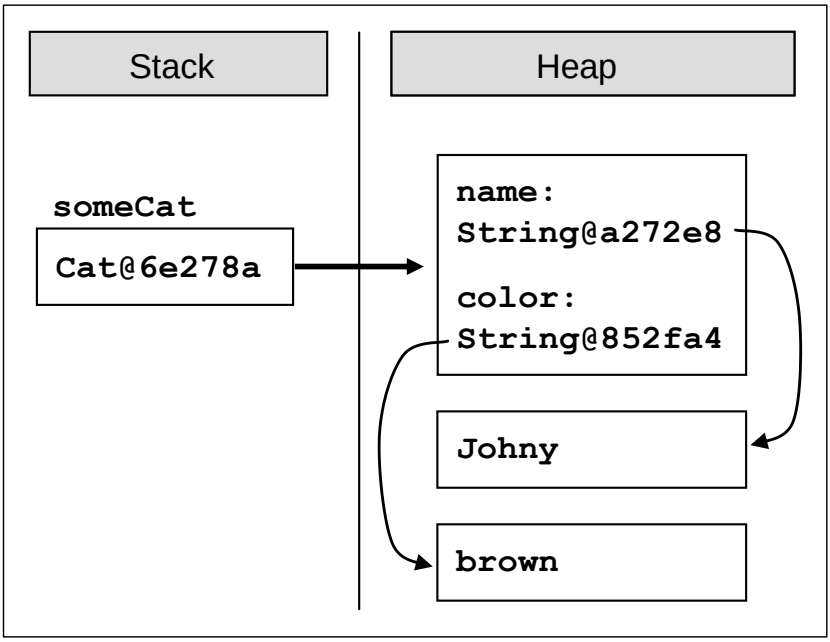
Сега ще разгледаме леко променен вариант на горния пример, при който задаваме параметри при създаването на обекта:

```
Cat myBrownCat = new Cat("Johnny", "brown");
```

В този случай искаме обектът **myBrownCat** да представлява котка, която се казва Johnny и има кафяв цвят. Указваме това чрез думите **"Johnny"** и **"brown"**, написани в скоби след името на класа.

При създаването на обект с оператора **new** се случват две неща: заделя се памет за този обект и се извършва начална инициализация на член-данните му. Инициализацията се осъществява от специален метод на класа, наречен **конструктор**. В горния пример инициализиращите параметри са всъщност

параметри на конструктора на класа. Ще се спрем по-подробно на конструкторите след малко. Понеже член-променливите **name** и **color** на класа Cat са от референтен тип (от класа **String**), те се записват също в динамичната памет (heap) и в самия обект стоят техните референции (адреси). Следващата картинка показва това нагледно:

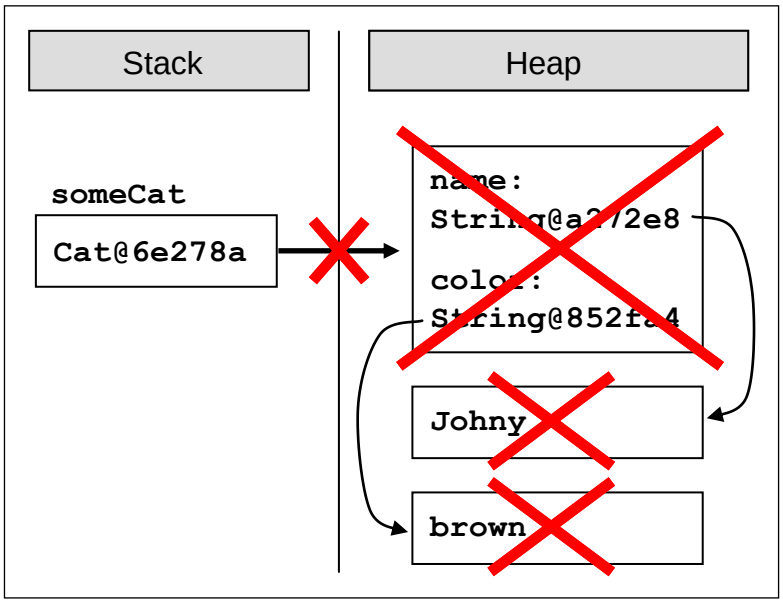


Освобождаване на обектите

Важна особеност на работата с обекти в Java е, че обикновено няма нужда от ръчното им разрушаване и освобождаване на паметта, заета от тях. Това е възможно поради наличието на **garbage collector** във виртуалната машина, който се грижи за това вместо нас. Обектите, към които в даден момент вече няма референция в програмата автоматично се унищожават и паметта, която заемат се освобождава. По този начин се предотвратяват много потенциални бъгове и проблеми. Ако искаме ръчно да освободим даден обект, трябва да унищожим референция към него, например така:

```
myBrownCat = null;
```

Това не унищожава обекта веднага, но го оставя в състояние, в което той е недостъпен от програмата и при следващото включване на системата за почистване на паметта (garbage collector) той ще бъде освободен:



Достъп до полета на обекта

Достъпът до полетата и свойствата (properties) на даден обект става чрез оператора **.** (точка), поставен между името на обекта и името на полето (или свойството). Операторът **.** не е необходим в случай, че достъпваме поле или свойство на даден клас в тялото на метод на същия клас.

Можем да достъпваме полетата и свойствата или с цел да извлечем данните от тях, или с цел да запишем нови данни. В случай на свойство, достъпът се реализира чрез два специални метода, наречени **getter** и **setter**. Те извършват съответно извличането на стойността на свойството и присвояването на нова стойност. В дефиницията на класа Cat (която дадохме по-горе) такива методи са **getName()** и **setName(…)**.

Достъп до полета на обекта – пример

Ще дадем прост пример за употребата на свойство на обект, като използваме вече дефинирания по-горе клас **Cat**. Създаваме инстанция **myCat** на класа **Cat** и присвояваме стойност **"Alfred"** на свойството **name.** След това извеждаме на стандартния изход форматиран низ с името на нашата котка. Следва реализацията на примера:

```
public class CatManipulating {
   public static void main(String[] args) {
      Cat myCat = new Cat();
      myCat.name = "Alfred";

      System.out.println("The name of my cat is %s.",myCat.name);
```

```
  }
}
```

Извикване на методи на обект

Извикването на методите на даден обект става отново чрез оператора **.** (точка). Операторът точка не е нужен единствено, в случай че съответният метод се извиква в тялото на друг метод на същия клас.

Тук е моментът да споменем факта, че методите на класовете имат **модификатори за достъп** **public**, **private** или **protected**, чрез които възможността за извикването им може да се ограничава. Ще разгледаме подробно тези модификатори в темата "Дефиниране на класове". Засега ще кажем само, че модификаторът за достъп **public** не въвежда никакво ограничение за извикването на съответния метод.

Извикване на методи на обект – пример

Ще допълним примера, който вече дадохме като извикаме метода **sayMiau** на класа **Cat**. Ето какво се получава:

```
public class CatManipulating {
   public static void main(String[] args) {
      Cat myCat = new Cat();
      myCat.name = "Alfred";

      System.out.println("The name of my cat is %s.%n",
        myCat.name);
      myCat.sayMiau();
   }
}
```

След изпълнението на горната програма на стандартния изход ще бъде изведен следния текст:

```
The name of my cat is Alfred.
Cat Alfred said: Miauuuuuu!
```

Конструктори

Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде със или без параметри. Конструктор без параметри наричаме още конструктор по подразбиране (default constructor).

Конструктори с параметри

Конструкторът може да имат параметри, както всеки друг метод. Всеки клас може да има произволен брой конструктори с единственото ограничение, че броят и типът на параметрите им трябва да бъде различен. При създаването на обект от този клас се извиква точно един от дефинираните конструктори.

При наличието на няколко конструктора в един клас естествено възниква въпросът кой от тях се извиква при създаването на обект. Този проблем се решава по много интуитивен начин. Подходящият конструктор се избира автоматично в зависимост от подадените параметри при създаването на обекта. Използва се принципът на най-добро съвпадение.

Извикване на конструктори – пример

Да разгледаме отново дефиницията на класа **Cat** и по-конкретно двата конструктора на класа:

```
public class Cat {
   // Field name
   private String name;
   // Field color
   private String color;

   ...

   // Default constructor
   public Cat() {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(String name, String color) {
      this.name = name;
      this.color = color;
   }

   ...

}
```

Ще използваме тези конструктори, за да илюстрираме употребата на конструктор без и с параметри. При така дефинирания клас **Cat** ще дадем пример за създаването на негови инстанции чрез всеки от двата конструктора. Единият обект ще бъде обикновена неопределена котка, а другият – нашата кафява котка Johnny. След това ще изпълним метода **sayMiau** на всяка от двете и ще разгледаме резултата. Следва изходният код:

```
public class CatManipulating {
   public static void main(String[] args) {
      Cat someCat = new Cat();

      someCat.sayMiau();
      System.out.println("The color of cat %s is %s.%n",
         someCat.name, someCat.color);

      Cat myBrownCat = new Cat("Johnny", "brown");

      myBrownCat.sayMiau();
      System.out.println("The color of cat %s is %s.%n",
         myBrownCat.name, myBrownCat.color);
}
```

В резултат от изпълнението на програмата се извежда следният текст на стандартния изход:

```
Cat Unnamed said: Miauuuuuu!
The color of cat Unnamed is gray.
Cat Johnny said: Miauuuuuu!
The color of cat Johnny is brown.
```

Статични полета и методи

Член-данните, които разглеждахме досега реализират състояния на обектите и са пряко свързани с конкретни инстанции на класовете. В ООП има специална категория полета и методи, които се асоциират с тип данни (клас), а не с конкретна инстанция (обект). Наричаме ги **статични членове** (static members), защото са независими от конкретните обекти. Нещо повече – те се използват, без да има създадена инстанция на класа, в който са дефинирани. Ще разгледаме накратко статичните членове в Java – това могат да бъдат полета, методи и конструктори.

Статично поле или метод се дефинира чрез ключовата дума **static**, поставена преди типа на полето или типа на връщаната стойност на метода. При дефинирането на статичен конструктор думата **static** се поставя преди името на конструктора. Статичните конструктори не са предмет на настоящата тема – засега ще се спрем на статичните полета и методи.

Кога да използваме статични полета и методи?

За да отговорим на този въпрос трябва преди всичко добре да разбираме разликата между статичните и нестатичните (non-static) членове. Ще разгледаме по-детайлно каква е тя.

Вече обяснихме основната разлика между двата вида членове. Нека интерпретираме класа като категория, а обекта като елемент, попадащ в

тази категория. Тогава статичните членове отразяват състояния и поведения на самата категория, а нестатичните – състояния и поведения на отделните елементи на категорията.

Сега ще обърнем по-специално внимание на инициализацията на статичните и нестатичните полета. Вече знаем, че нестатичните полета се инициализират заедно с извикването на конструктор на класа при създаването на негова инстанция – или в тялото на конструктора, или извън него. Инициализацията на статичните полета, обаче, не може да става при създаването на обект от класа, защото те могат да бъдат използвани, без да има създадена инстанция на този клас. Важно е да се знае следното:

Статичните полета се инициализират, когато типът данни (класът) се използва за пръв път по време на изпълнението на програмата.

Време е да видим как се използват статични полета и методи на практика.

Статични полета и методи – пример

Примерът, който ще дадем решава следната проста задача: нужен ни е метод, който всеки път връща стойност с едно по-голяма от стойността, върната при предишното извикване на метода. Избираме първата върната от метода стойност да бъде 0. Очевидно такъв метод генерира редицата на естествените числа. Подобна функционалност има широко приложение в практиката – за еднозначно номериране на обекти. Сега ще видим как може да се реализира с инструментите на ООП.

Да приемем, че методът е наречен **`nextValue()`** и е дефиниран в клас с име **`Sequence`**. Класът има поле **`currentValue`** от тип **`int`**, което съдържа последно върнатата стойност от метода. Искаме в тялото на метода да се извършват последователно следните две действия: да се увеличава стойността на полето и да се връща като резултат новата му стойност. Връщаната от метода стойност очевидно не зависи от конкретна инстанция на класа **`Sequence`**. Поради тази причина методът и полето са статични. Следва описаната реализация на класа:

```
public class Sequence {
   // Static field
   private static int currentValue = -1;

   // Intentionally deny instantiation of this class
   private Sequence() {
   }

   // Static method
   public static int nextValue() {
      currentValue++;
```

```
        return currentValue;
    }
}
```

Наблюдателният читател е забелязал, че така дефинираният клас има конструктор по подразбиране, който е деклариран като **private**. Тази употреба на конструктор може да изглежда особена, но е съвсем умишлена. Добре е да знаем следното:

Клас, който има само private конструктори не може да бъде инстанциран. Такъв клас обикновено има само статични членове и се нарича utility клас.

Засега няма да навлизаме в детайли за употребата на **модификаторите за достъп** **public**, **private** и **protected**. Ще ги разгледаме подробно в главата "Дефиниране на класове".

Нека сега видим една проста програма, която използва класа **Sequence**:

```
public class SequenceManipulating {
   public static void main(String[] args) {
      System.out.printf("Sequence[1..3]: %d, %d, %d%n",
         Sequence.nextValue(), Sequence.nextValue(),
         Sequence.nextValue());
}
```

Извеждаме на стандартния изход първите три естествени числа чрез последователно извикване на метода **nextValue() на класа Sequence.** Резултатът от този код е следният:

```
Sequence[1..3]: 0, 1, 2
```

Примери за системни Java класове

След като вече се запознахме с основната функционалност на обектите, ще разгледаме накратко няколко често използвани системни класа от стандартните библиотеки на Java. По този начин ще видим на практика казаното дотук, а също ще покажем как системните класове улесняват работата ни.

Класът System

Започваме с един от основните системни класове в Java. Той съдържа набор от полезни полета и методи, улесняващи взаимодействието на програмите с операционната система. Ето част от функционалността, която предоставя този клас:

- Стандартните входно-изходни потоци **System.out**, **System.in** и **System.err** (които вече сме разглеждали).
- Достъп до външно дефинирани свойства (properties) и променливи на обкръжението (environment variables), които няма да разглеждаме в настоящата книга.
- Средства за зареждане на файлове и библиотеки.

Сега ще покажем едно интересно приложение на метод на класа System, което често се използва в практиката при разработката на програми с критично бързодействие. Ще засечем времето за изпълнение на фрагмент от изходния код с помощта на метода **currentTimeMillis()**. Ето как става това:

```
public class SystemTest {
   public static void main(String[] args) {
      int sum = 0;
      long startTime = System.currentTimeMillis();

      // The code fragment to be tested
      for(int i = 0; i < 10000000; i++) {
         sum++;
      }

      long endTime = System.currentTimeMillis();
      System.out.printf("The time elapsed is %f sec",
         (endTime - startTime)/1000.0);
   }
}
```

Методът **currentTimeMillis()** връща като резултат броя милисекунди, които са изтекли от 0:00 часа на 1 януари 1970 година до момента на извикването на метода. С негова помощ засичаме изтеклите милисекунди преди и след изпълнението на критичния код. Тяхната разлика е всъщност търсеното време за изпълнение на фрагмента код, измерено в милисекунди.

В резултат от изпълнението на програмата на стандартния изход се извежда резултат от следния вид (засеченото време варира в зависимост от конкретната компютърна конфигурация и нейното натоварване):

```
The time elapsed is 0,016000 sec
```

Класът String

Вече сме споменавали класа в Java, който представя символни низове (последователности от символи). Да припомним, че можем да считаме низовете за примитивен тип данни в Java, въпреки че работата с тях се различава до известна степен от работата с другите примитивни типове

(цели и реални числа, булеви променливи и др.). Ще се спрем по-подробно на тях в темата "Символни низове".

Класът Math

Съдържа методи за извършването на основни числови операции като повдигане в степен, логаритмуване, коренуване и тригонометрични функции. Ще дадем един прост пример, който илюстрира употребата му.

Съставяме програма, която пресмята лицето на триъгълник по дадени дължини на две от страните и ъгъла между тях в градуси. За тази цел имаме нужда от методите **sin(…)** и **toRadians(…)** на класа **Math**. Следва примерна реализация:

```
public class MathTest {
   public static void main(String[] args) {
      java.util.Scanner input = new java.util.Scanner(System.in);

      System.out.println("Length of the first side:");
      double a = input.nextDouble();
      System.out.println("Length of the second side:");
      double b = input.nextDouble();
      System.out.println("Size of the angle in degrees:");
      int angle = input.nextInt();

      System.out.printf("Face of the triangle: %f%n",
         0.5 * a * b * Math.sin(Math.toRadians(angle)));
   }
}
```

Можем лесно да тестваме програмата като проверим дали пресмята правилно лицето на равностранен триъгълник. За допълнително улеснение избираме дължина на страната да бъде 2 – тогава лицето му намираме с добре известната формула:

$$S = \frac{\sqrt{3}}{4} 2^2 = \sqrt{3} = 1{,}7320508...$$

Въвеждаме последователно числата 2, 2, 60 и на стандартния изход се извежда:

```
Face of the triangle: 1,732051
```

Класът Math – още примери

Освен математически методи класът **Math** дефинира и две добре известни в математиката константи: числото π и Неперовото число ***e***. Ето как се достъпват те:

```
System.out.println(Math.PI);
System.out.println(Math.E);
```

При изпълнение на горния код се получава следния резултат:

```
3.141592653589793
2.718281828459045
```

Класът Random

Понякога в програмирането се налага да използваме случайни числа. Например искаме да генерираме 6 случайни числа в интервала между 1 и 49 (ТОТО 6/49). Това можем да направим използвайки класа **java.util.Random** и неговия метод **nextInt()**. Преди използваме класа **Random** трябва да създадем негова инстанция, при което тя се инициализира със случайна стойност (извлечена от текущото системно време в операционната система). След това можем да генерира случайно число в интервала **[0…n)** чрез извикване на метода **nextInt(n)**. Забележете, че този метод може да върне нула, но връща винаги случайно число по-малко от зададената стойност **n**. Затова ако искаме да получим число в интервала **[0…49]**, трябва използваме израза **nextInt(49)+1**. Ето сорс кода на една програма, която използвайки класа **Random** генерира 6 случайни числа от ТОТО 6/49:

```
import java.util.Random;

public class TOTO649 {
   public static void main(String[] args) {
      Random rand = new Random();
      for (int number=1; number<=6; number++) {
         int randomNumber = rand.nextInt(49) + 1;
         System.out.printf("%d ", randomNumber);
      }
   }
}
```

Ето как изглежда един възможен изход от работата на програмата:

```
14 49 7 16 29 2
```

Класът Random – още един пример

За да ви покажем колко полезен може да е генераторът на случайни числа в Java, ще си поставим за задача да генерираме случайна парола, която е дълга между 8 и 15 символа съдържа поне две главни букви, поне две малки букви, поне една цифра и поне три специални знака. За целта ще използваме следния алгоритъма:

1. Започваме от празна парола. Създаваме генератор на случайни числа.
2. Генерираме два пъти по една случайна главна буква и я поставяме на случайна позиция в паролата.
3. Генерираме два пъти по една случайна малка буква и я поставяме на случайна позиция в паролата.
4. Генерираме една случайна цифра и я поставяме на случайна позиция в паролата.
5. Генерираме три пъти по един случаен специален символ и го поставяме на случайна позиция в паролата.
6. До момента паролата трябва да се състои от 8 знака. За да я допълним до най-много 15 пъти можем случаен брой пъти (между 0 и 7) да вмъкнем на случайна позиция в паролата случаен знак (главна буква или малка буква или цифра или специален символ).

Следва имплементация на алгоритъма:

```
import java.util.Random;

public class RandomPasswordGenerator {

   private static final String CAPITAL_LETTERS =
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   private static final String SMALL_LETTERS =
      "abcdefghijklmnopqrstuvwxyz";
   private static final String DIGITS = "0123456789";
   private static final String SPECIAL_CHARS =
      "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
   private static final String ALL_CHARS =
      CAPITAL_LETTERS + SMALL_LETTERS + DIGITS + SPECIAL_CHARS;

   private static Random rnd = new Random();

   public static void main(String[] args) {
      StringBuilder password = new StringBuilder();

      // Generate two random capital letters
      for (int i=1; i<=2; i++) {
         char capitalLetter = generateChar(CAPITAL_LETTERS);
         insertAtRandomPosition(password, capitalLetter);
      }

      // Generate two random small letters
      for (int i=1; i<=2; i++) {
         char smallLetter = generateChar(SMALL_LETTERS);
         insertAtRandomPosition(password, smallLetter);
```

```
      }

      // Generate one random digit
      char digit = generateChar(DIGITS);
      insertAtRandomPosition(password, digit);

      // Generate 3 special characters
      for (int i=1; i<=3; i++) {
         char specialChar = generateChar(SPECIAL_CHARS);
         insertAtRandomPosition(password, specialChar);
      }

      // Generate few random characters (between 0 and 7)
      int count = rnd.nextInt(8);
      for (int i=1; i<=count; i++) {
         char specialChar = generateChar(ALL_CHARS);
         insertAtRandomPosition(password, specialChar);
      }

      System.out.println(password);
   }

   private static void insertAtRandomPosition(
         StringBuilder password, char character) {
      int randomPosition = rnd.nextInt(password.length()+1);
      password.insert(randomPosition, character);
   }

   private static char generateChar(String availableChars) {
      int randomIndex = rnd.nextInt(availableChars.length());
      char randomChar = availableChars.charAt(randomIndex);
      return randomChar;
   }
}
```

Нека обясним някои неясни моменти в сорс кода. Да започнем от дефинициите на константи:

```
private static final String CAPITAL_LETTERS =
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private static final String SMALL_LETTERS =
   "abcdefghijklmnopqrstuvwxyz";
private static final String DIGITS = "0123456789";
private static final String SPECIAL_CHARS =
   "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
private static final String ALL_CHARS =
   CAPITAL_LETTERS + SMALL_LETTERS + DIGITS + SPECIAL_CHARS;
```

Константите в Java представляват неизменими променливи, чиито стойности се задават по време на инициализацията им в сорс кода на програмата и след това не могат да бъдат променяни. Те се декларират с модификаторите **`static`** и **`final`**. Използват се за дефиниране на дадено число или стринг, което се използва след това многократно в програмата. По този начин се спестяват повторенията на определени стойности в сорс кода и се позволява лесно тези стойности да се променят като се бута само на едно място в сорс кода. Например ако в даден момент решим, че символът "**`,`**" (запетая) не трябва да се ползва при генерирането на пароли, можем да променим само 1 ред в програмата (съответната константа) и промяната ще се отрази навсякъде, където е използвана съответната константа. Константите в Java се изписват само с главни букви, като за разделител между думите се ползва символът "**`_`**" (долна черта).

Нека обясним и как работят останалите части от програмата. В началото като статична член-променлива в класа **`RandomPasswordGenerator`** се създава генераторът на случайни числа **`rnd`**. Понеже тази променлива **`rnd`** е дефинирана в самия клас (не в **`main()`** метода), тя е достъпна от целия клас (от всички негови методи) и понеже е обявена за статична, тя е достъпна и от статичните методи. По този навсякъде, където програмата има нужда от случайна целочислена стойност, се използва един и същ генератор на случайни числа, който се инициализира при зареждането на класа **`RandomPasswordGenerator`**.

Методът **`generateChar()`** връща случайно избран символ измежду множество символи, подадени му като параметър. Той работи много просто: избира случайна позиция в множеството символи и връща символът на тази позиция.

Методът **`insertAtRandomPosition()`** също не е сложен. Той избира случайна позиция в **`StringBuilder`** обекта, който му е подаден и вмъква на тази позиция подадени символ.

Ето примерен изход от програмата за генериране на пароли, която разгледахме и обяснихме как работи:

```
8p#Rv*yTl{tN4
```

Пакети

Пакет (package/namespace/context) в ООП наричаме абстрактен контейнер за група класове, които са обединени от общ признак или се използват в общ контекст. Пакетите спомагат за една по-добра логическа организация на изходния код. Създават семантично разделение на класовете в категории и улесняват употребата им в програмния код. Сега ще се спрем на пакетите в Java и ще видим как можем да ги използваме.

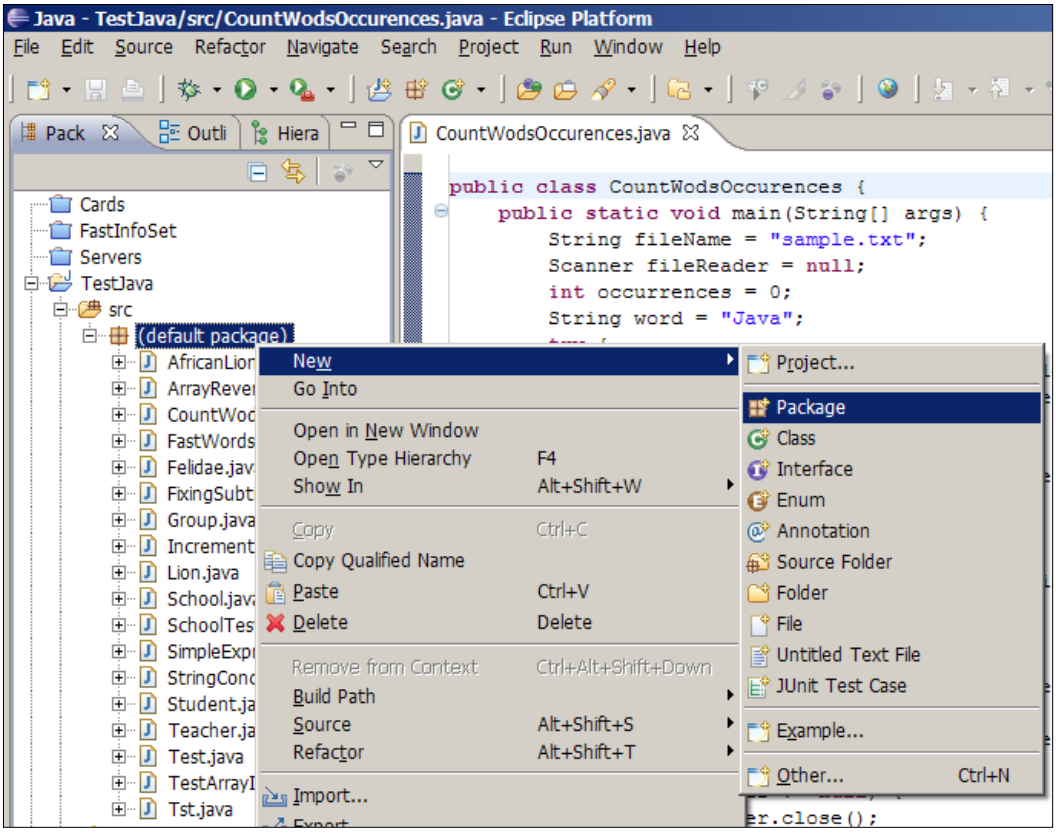
Какво представляват пакетите в Java?

Пакетите (packages) в Java представляват именувани групи класове, които са логически свързани и се съхраняват в отделни файлове в една и съща директория във файловата система. Прието е името на папката да съвпада с името на пакета и имената на файловете да съвпадат с имената на класовете, които се съхраняват в тях. Трябва да отбележим, че в някои езици за програмиране компилацията на изходния код на даден пакет е независима от разпределението на елементите на пакета в папки и файлове на диска. В Java, обаче, така описаната файлова организация на пакетите е напълно задължителна (ако не е спазена, възниква грешка при компилацията).

Нека сега разгледаме механизма за дефиниране на пакети.

Дефиниране на пакети

В случай, че искаме да създаден нов пакет или да създадем нов клас, който ще принадлежи на даден пакет, за целта в Eclipse има удобни команди в контекстното меню на Package Explorer (при щракане с десния бутон на мишката върху съответната папка):



Package Explorer по подразбиране се визуализира като страница в лявата част на интегрираната среда. Ще покажем нагледно как можем да добавим нов клас към вече съществуващия пакет **myPackage** чрез контекстното меню на Package Explorer в Eclipse.

Ако сме дефинирали клас в собствен файл и искаме да го добавим към нов или вече съществуващ пакет, не е трудно да го направим ръчно. Достатъчно е да преместим файла в папката на пакета и да добавим следния ред в началото на файла:

```
package <package_name>;
```

При дефиницията използваме ключовата дума **package**, последвана от пълното име на пакета. Прието е имената на пакетите в Java да започват с малка буква и да бъдат изписвани в camelCase. Например, ако трябва да направим пакет, който съдържа помощни функции за работа със символни низове, можем да го именуваме, **stringUtils**, а не **StringUtils**.

Вложени пакети

Освен класове, пакетите могат да съдържат в себе си и други пакети. По този начин съвсем интуитивно се изгражда йерархия от пакети, която позволява още по-прецизно разделение на класовете според тяхната семантика.

При назоваването на пакетите в йерархията се използва символът **.** за разделител (точкова нотация). Например пакетът **java.lang** съдържа пакета **reflect** и така пълното название на вложения пакет **reflect** е **java.lang.reflect**.

Пълни имена на класовете

За да разберем напълно смисъла на пакетите, важно е да знаем следното:

Класовете трябва да имат уникални имена само в рамките на пакета, в който са дефинирани. Имената на класовете извън него са произволни.

Извън даден пакет наистина може да има класове с произволни имена, без значение дали съвпадат с някои от имената на класовете в пакета. Това е така, защото класовете в пакета са определени еднозначно от неговия контекст. Време е да видим как се определя синтактично тази еднозначност.

Пълно име на клас наричаме собственото име на класа, предшествано от името на пакета, в който този клас е дефиниран. Пълното име на всеки клас е уникално. Ще отбележим, че тук отново използваме точковата нотация:

```
<package_name>.<class_name>
```

Нека вземем за пример системния клас **Array**, дефиниран в пакета **java.lang.reflect** (вече сме го споменавали като пример за системен клас). Съгласно дадената дефиниция, пълното име на този клас е **java.lang.reflect.Array**.

Включване на пакет

При изграждането на приложения в зависимост от предметната област често се налага многократното използване на два или повече класа от един пакет. За удобство на програмиста има механизъм за **включване** на пакет към текущото приложение, което става по време на компилацията му. По този начин могат свободно да се използват всички класове, дефинирани в пакета, без да е необходимо използването на техните пълни имена.

Включването на пакет към файл с изходен код се извършва чрез ключовата дума **import** по следния начин:

```
import <package_name>.*;
```

Ще обърнем внимание на една важна особеност при включването на пакети по показания начин. Символът * означава, че включваме всички класове, които се съдържат в пакета **<package_name>**, но трябва да знаем следното:

По този начин не се включват класовете от вложените пакети на пакета, който включваме.

Например включването на пакета **java.*** не включва класовете, съдържащи се в пакета **java.io.***. При употребата им трябва да ги назоваваме с пълните им имена.

Включване на пакет – пример

За да илюстрираме принципа на включването на пакет, ще разгледаме следната програма:

```
public class PackageImportTest {
   public static void main(String[] args) {
      java.util.Scanner input = new java.util.Scanner(System.in);
      java.util.ArrayList<Integer> ints = new
         java.util.ArrayList<Integer>();
      java.util.ArrayList<Double> doubles = new
         java.util.ArrayList<Double>();

      while(true) {
         System.out.println("Enter an int or a double:");
```

```
        if(input.hasNextInt()) {
           ints.add(input.nextInt());
        } else if(input.hasNextDouble()) {
           doubles.add(input.nextDouble());
        } else {
           break;
        }
     }

     System.out.printf("You entered these ints: %s%n",
        ints.toString());
     System.out.printf("You entered these doubles: %s%n",
        doubles.toString());
   }
}
```

Нека сега видим как работи горната програма: въвеждаме последователно стойностите **4**, **1,53**, **0,26**, **7**, **2**, **end**. Получаваме следния резултат на стандартния изход:

```
You entered these ints: [4, 7, 2]
You entered these doubles: [1.53, 0.26]
```

Програмата извършва следната дейност: дава на потребителя възможност да въвежда последователно числа, които могат да бъдат цели или реални. Въвеждането продължава до момента, в който бъде въведена стойност, различна от число. След това на стандартния изход се извеждат два списъка съответно с целите и с реалните въведени числа.

За реализацията на описаните действия използваме три помощни обекта съответно от тип **java.util.Scanner**, **java.util.ArrayList<Integer>** и **java.util.ArrayList<Double>**. Очевидно е, че пълните имена на класовете правят кода непрегледен и създават неудобство при употребата си. Можем лесно да избегнем този ефект като включим пакета **java.util** и използваме директно собствените имена на класовете. Следва промененият вариант на горната програма:

```
import java.util.*;

public class PackageImportTest {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      ArrayList<Integer> ints = new ArrayList<Integer>();
      ArrayList<Double> doubles = new ArrayList<Double>();

      while(true) {
```

```
        System.out.println("Enter an int or a double:");

        if(input.hasNextInt()) {
           ints.add(input.nextInt());
        } else if(input.hasNextDouble()) {
           doubles.add(input.nextDouble());
        } else {
           break;
        }
     }

     System.out.printf("You entered these ints: %s%n",
        ints.toString());
     System.out.printf("You entered these doubles: %s%n",
        doubles.toString());
   }
}
```

Упражнения

1. Напишете програма, която извежда на стандартния изход броя на дните, часовете и минутите, които са изтекли от 1 януари 1970 година до момента на изпълнението на програмата. За реализацията използвайте класа **System**.

2. Напишете програма, която по дадени два катета намира хипотенузата на правоъгълен триъгълник. Реализирайте въвеждане на дължините на катетите от стандартния вход, а за пресмятането на хипотенузата използвайте методи на класа **Math**.

3. Дефинирайте свой собствен пакет **chapter10** и поставете в него двата класа **Cat** и **Sequence**, които използвахме в примерите на текущата тема. Направете още един собствен пакет с име **chapter10.examples** и в него направете клас, който извиква класовете **Cat** и **Sequence**.

4. Напишете програма, която създава 10 обекта от тип **Cat**, дава им имена от вида **CatN**, където **N** е уникален пореден номер на обекта, и накрая извиква метода **sayMiau()** на всеки от тях. За реализацията използвайте вече дефинирания пакет **chapter10**.

5. Напишете програма, която генерира случайно рекламно съобщение за някакъв продукт. Съобщенията трябва да се състоят от хвалебствена фраза, следвани от хвалебствена случка, следвани от автор (първо и второ име) и град, които се избират от предварително подготвени списъци. Например, нека имаме следните списъци:

   - Хвалебствени фрази: {"Продуктът е отличен.", "Това е страхотен продукт.", "Постоянно ползвам този продукт.", "Това е най-добрият продукт от тази категория."}.

- Хвалебствени случки: {"Вече се чувствам добре.", "Успях да се променя.", "Той направи чудо.", "Не мога да повярвам, но вече се чувствам страхотно.", "Опитайте и вие. Аз съм много доволна."}.
- Първо име на автор: {"Диана", "Петя", "Стела", "Елена", "Катя"}.
- Второ име на автор: {"Иванова", "Петрова", "Кирова"}.
- Градове: {"София", "Пловдив", "Варна", "Русе", "Бургас"}.

Тогава програма би могла да изведе следното случайно-генерирано рекламно съобщение:

```
Постоянно ползвам този продукт. Опитайте и вие. Аз съм доволна. -
– Елена Петрова, Варна
```

6. Ако често ругаете компютъра, можете да развиете идеята от предишната задача и да напишете програма, която генерира цветущи опашати ругатни.

Решения и упътвания

1. Използвайте метода **`System.currentTimeMillis()`**, за да получите броя на изтеклите милисекунди. Използвайте факта, че в една секунда има 1000 милисекунди и пресметнете минутите, часовете и дните.
2. Хипотенузата на правоъгълен триъгълник се намира с помощта на известната теорема на Питагор: $\mathbf{a}^2 + \mathbf{b}^2 = \mathbf{c}^2$, където **a** и **b** са двата катета, а **c** е хипотенузата. Коренувайте двете страни, за да получите формула за дължината на хипотенузата. За реализацията на коренуването използвайте метода **`sqrt(…)`** на класа **`Math`**.
3. Създайте нов проект в Eclipse, щракнете с десния бутон върху папката **`src`** и изберете от контекстното меню New -> Package. След като въведете име на пакета и натиснете [Finish], щракнете с десния бутон върху папката на новосъздадения пакет и изберете New -> Class. За име на новия клас въведете **`Cat`** и натиснете [Finish]. Подменете дефиницията на новосъздадения клас с дефиницията, която дадохме в тази тема. Направете същото за класа **`Sequence`**.
4. Създайте масив с 10 елемента от тип **`Cat`**. Създайте в цикъл 10 обекта от тип Cat (използвайте конструктор с параметри), като ги присвоявате на съответните елементи от масива. За поредния номер на обектите използвайте метода **`nextValue()`** на класа **`Sequence`**. Накрая отново в цикъл изпълнете метода **`sayMiau()`** на всеки от елементите на масива.
5. Използвайте класа **`java.util.Random`** и неговия метод **`nextInt()`**.
6. Първо дефинирайте граматика, която описва една ругатня. Примерно може да ползвате следната граматика:

   `ругатня = действие + допълнение към действието + обръщение`

```
действие = {"Ще те ... в", "Ще ти го ... в", … }
допълнение към действието = {"ушите", "устата", "носа", …}
обръщение = единично общение или
            епитет + единично общение или
            епитет + епитет + единично общение
единично общение = {"глупак", "простак", "идиот", …}
епитет = {"смахнат", "смешен", "гламав", …}
```

За всеки елемент от граматиката напишете по един метод, който генерира съответния елемент по случаен начин. За по-интересни резултати дефинирайте достатъчно дълги списъци с цветущи думички и фрази. Ако вложите малко иновация и усложните граматиката, ще се забавлявате дни наред!

Глава 12. Обработка на изключения

Автор

Лъчезар Цеков

Михаил Стойнов

Светлин Наков

В тази тема...

В настоящата тема ще се запознаем с изключенията в Java и обектно-ориентираното програмиране. Ще се научим как да ги прихващаме чрез конструкцията `try-catch`, как да ги предаваме на предходните методи чрез `throws` и как да хвърляме собствени или прихванати изключения. Ще дадем редица примери за използването им.

Ще разгледаме типовете изключения и йерархията, която образуват. Накрая ще се запознаем с предимствата при използването на изключения и с това как най-правилно да ги прилагаме в конкретни ситуации.

Какво е изключение?

Докато програмираме ние описваме постъпково какво трябва да направи компютъра и в повечето случаи разчитаме на нормалното изпълнение на програмата. В повече от 99% от времето програмите следват този нормален ход на изпълнение, но съществуват и изключения от това правило. Да речем, че искаме да прочетем файл и да покажем съдържанието му на екрана. Името на файла се подава от потребителя. По невнимание потребителя въвежда име на файл, който не съществува. Програмата няма да може да се изпълни нормално и да покаже съдържанието на файла на екрана. В този случай имаме изключение от правилното изпълнение на програмата и за него трябва да се сигнализира на потребителя и/или администратора.

Изключение

Изключение (exception) в общия случай е уведомление за дадено събитие, нарушаващо нормалната работа на една програма. Изключенията

дават възможност това необичайно събитие да бъде обработено и програмата да реагира по някакъв начин. Когато възникне изключение конкретното състояние на програмата се запазва и се търси **обработчик на изключението (exception handler)**.

Изключенията се предизвикват или **"хвърлят" (throw an exception)**.

Прихващане и обработка на изключения

Exception handling (**инфраструктура за обработка на изключенията**) е част от средата – механизъм, който позволява хвърлянето и прихващането на изключения. Част от тази инфраструктура са дефинираните езиковите конструкции за хвърляне и прихващане на изключения. Тя се грижи и затова изключението да стигне до кода, който може да го обработи.

Изключенията в ООП

В обектно-ориентираното програмиране (ООП) изключенията представляват мощно средство за **централизирана обработка на грешки** и изключителни (необичайни) ситуации. Те заместват в голяма степен процедурно-ориентирания подход, при който всяка функция връща като резултат от изпълнението си код на грешка (или неутрална стойност, ако не е настъпила грешка).

В ООП кодът, който извършва дадена операция, обикновено предизвиква изключение, когато в него възникне проблем и операцията не може да бъде изпълнена успешно. Методът, който извиква операцията може да прихване изключението и да обработи грешката или да пропусне изключението и да остави то да бъде прихванато от извикващият го метод. Така не е задължително грешките да бъдат обработвани непосредствено от извикващия код, а могат да се оставят за тези, които са го извикали. Това дава възможност управлението на грешките и необичайните ситуации да се извършва на много нива.

Друга основна концепция при изключенията е тяхната **йерархична същност**. Изключенията в ООП са класове и като такива могат да образуват йерархии посредством наследяване. При прихващането на изключения може да се обработват наведнъж цял клас от грешки, а не само дадена определена грешка (както е в процедурното програмиране).

В ООП се препоръчва чрез изключения да се управлява всяко състояние на грешка или неочаквано поведение, възникнало по време на изпълнението на една програма.

Кое е очаквано и кое неочаквано събитие е описано към края на тази глава.

Изключенията в Java

Изключение (exception) в Java представлява събитие, което уведомява програмиста, че е възникнало обстоятелство (грешка) непредвидено в нормалния ход на програмата. Това става като методът, в който е възникнала грешката изхвърля специален обект съдържащ информация за вида на грешката, мястото в програмата, където е възникнала, и състоянието на програмата в момента на възникване на грешката.

Всяко изключение в Java носи т.нар **stack trace** (няма да се мъчим да го превеждаме) – информация за това къде точно в кода е възникнала грешката. Ще го дискутираме подробно малко по-късно.

Пример за код, който хвърля изключения

Типичен пример за код, който хвърля изключения е следният метод:

```
public static void readFile(String fileName) {
   FileInputStream fis = new FileInputStream(fileName);
   BufferedReader in = new BufferedReader(
      new InputStreamReader(fis));
   String tmp = null;
   while ((tmp = in.readLine()) != null) {
      System.out.println(tmp);
   }
   in.close();
   fis.close();
}
```

Това е код, който се опитва да отвори текстов файл и да чете от него ред по ред докато файлът свърши. Повече за работата с файлове ще научите в главата "Текстови файлове". За момента, нека се съсредоточим не в класовете и методите за работа с файлове, в конструкциите за работа с изключения.

Подчертаните методи и конструктори са тези, в които се хвърлят изключенията. В примера конструкторът **`FileInputStream(fileName)`** хвърля **`FileNotFoundException`**, ако не съществува файл с име, каквото му се подава. Методите на потоците **`readLine()`** и **`close()`**, хвърлят **`IOException`** ако възникне неочакван проблем при входно-изходните операции. Този пример няма да се компилира (местата, където са грешките от компилация, са подчертани), защото хвърляните изключения трябва да бъдат прихванати и да бъдат подходящо обработени.

Как работят изключенията?

Ако по време на нормалния ход на програмата някой от извикваните методи неочаквано хвърли изключение, то нормалният ход на програмата се преустановява. Това ще се случи, ако например възникне изключение от

типа **FileNotFoundException** при инициализиране на файловия поток от горния пример. Нека разгледаме сления ред:

```
FileInputStream fis = new FileInputStream(fileName);
```

Ако се случи изключение, променливата **fis** няма да бъде инициализирана и ще остане със стойност **null**. Нито един от следващите редове от метода няма да бъде изпълнен. Програмата ще преустанови своя ход докато виртуалната машина не намери обработчик на възникналото изключение **FileNotFoundException**.

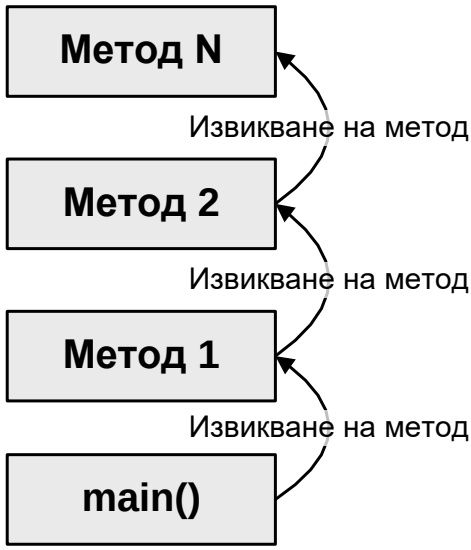
Прихващане на изключения в Java

След като един метод хвърли изключение, виртуалната машина търси код, който да го прихване и евентуално обработи. За да разберем как действа този механизъм ще разгледаме понятието **стек** на извикване на методите. Това е същият този стек, в който се записват всички променливи в програмата, параметрите на методите и стойностните типове.

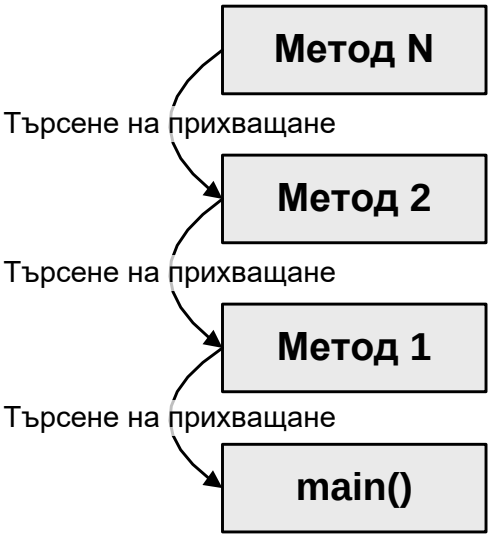
Всяка програма на Java започва с **main()** метод. В него може да се извика друг метод да го наречем "Метод 1", който от своя страна извиква "Метод 2" и т.н., докато се извика "Метод N".

Когато "Метод N" свърши работата си управлението на програмата се връща на предходния и т. н., докато се стигне до **main()** метода. След като се излезе от него свършва и програмата. Като се извиква нов метод той се добавя най-отгоре в стека, а като свърши изпълнението му метода се изважда от стека.

Можем да визуализираме този процес на извикване на методите един от друг по следния начин:



Процесът на търсене и прихващане на изключение е обратният на този за извикване на методи. Започва се от метода, в който е възникнало изключението и се върви в обратна посока докато се намери метод, където изключението е прихванато:



Не всички изключения трябва да се прихващат, но затова ще стане дума след малко в частта за видовете изключения.

Програмна конструкция try-catch

За да прихванем изключение обгръщаме парчето код, където може да възникне изключение с програмната конструкция **try-catch**:

```
try {
   Some sode that may throw and exception
} catch (ExceptionType objectName) {
   // Code handling an Exception
} catch (ExceptionType objectName) {
   // Code handling an Exception
}
```

Конструкцията се състои от един **try** блок, обгръщащ валидни Java конструкции, които могат да хвърлят изключения, следван от един или много **catch** блока, които обработват различни по тип изключения. В **catch** блокът **ExceptionType** трябва да е тип на клас, който е наследник на класа **java.lang.Throwable**. В противен случай ще получим проблем при компилация. Изразът в скобите след **catch** играе роля на декларация на променлива и затова вътре в блока **catch** можем да използваме **objectName**, за да извикваме методите или да използваме свойствата на изключението.

Прихващане на изключения – пример

Да направим така, че горният пример да се компилира. Заграждаме целият проблемен код, където могат да се хвърлят изключения с **try-catch** блок и добавяме прихващане на двата вида изключения:

```
public static void readFile(String fileName) {
   try {
      // Exceptions could be thrown below
      FileInputStream fis = new FileInputStream(fileName);
      BufferedReader in = new BufferedReader(
         new InputStreamReader(fis));
      String tmp = null;
      while ((tmp = in.readLine()) != null) {
         System.out.println(tmp);
      }
      in.close();
      fis.close();
   } catch (FileNotFoundException e) {
      // Exception handler for FileNotFoundException
      // We just inform the client that there is no such file
      System.out.println("The file \"" + fileName +
         "\" does not exist! Unable to read it.");
   } catch (IOException e) {
      // Exception handler for IOException
      e.printStackTrace();
   }
}
```

Добре, сега методът се компилира, но работи по малко по-различен начин. При възникване на **FileNotFoundException** по време на изпълнението на **new FileInputStream(fileName)** виртуалната машина няма да изпълни следващите редове, а ще се прескочи чак на реда, където изключението е прихванато с **catch (FileNotFoundException e)** и ще се изпълни блока след него:

```
catch (FileNotFoundException e) {
   // Exception handler for FileNotFoundException
   // We just inform the client that there is no such file
   System.out.println("The file \"" + fileName +
      "\" does not exist! Unable to read it.");
}
```

Като обработка на изключението просто потребителите ще бъдат информирани, че такъв файл не съществува. Това се извършва чрез съобщение, изведено на стандартния изход.

Аналогично, ако възникне изключение от тип **IOException** по време на изпълнението на метода **in.readLine()**, то се обработва от блока:

```
catch (IOException e) {
   // Exception handler for IOException
   e.printStackTrace();
}
```

Понеже не знаем естеството на грешката, породила грешно четене, отпечатваме цялата информация за изключението на стандартния изход.

Редовете код между мястото на възникване на изключението и мястото на прихващане и обработка не се изпълняват.

Отпечатването на цялата информация от изключението (stack trace) на потребителя не винаги е добра практика! Как най-правилно се обработват изключения е описано в частта за добри практики.

С така прихванати изключения примерът вече се компилира и може да се изпълни посредством **main()** метод, подобен на следния:

```
public static void main(String[] args) {
   readFile("C:\\tools\\eclipse\\eclipse.ini");
}
```

Да обединим двата метода в клас и да се опитаме да го изпълним като подаваме на метода **readFile()** първо съществуващ, а после и липсващ текстов файл и да видим какво ще получим. Ето го класът:

ReadFileExample**.java**

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReadFileExample {
   public static void readFile(String fileName) {
      try {
         // Exceptions could be thrown below
         FileInputStream fis = new FileInputStream(fileName);
         BufferedReader in = new BufferedReader(
           new InputStreamReader(fis));
         String line = null;
         while ((line = in.readLine()) != null) {
            System.out.println(line);
```

```
            }
            in.close();
            fis.close();
        } catch (FileNotFoundException e) {
            // Exception handler for FileNotFoundException
            // We just inform the client that there is
            // not such file
            System.out.println("The file \"" + fileName +
              "\" does not exist! Unable to read it.");
        } catch (IOException e) {
            // Exception handler for IOException
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        readFile("C:\\tools\\eclipse\\eclipse.ini");
    }
}
```

Ако в **main()** метода подадем път до съществуващ текстов файл то съдържанието му ще бъде отпечатано на екрана. От друга страна, ако файлът е несъществуващ, то програмата дори няма да направи опит да го чете, а ще изведе съобщение за грешка от типа:

```
The file "C:\tools\eclipse\eclipse.ini" does not exist! Unable to
read it.
```

Ако възникне грешка при самото четене на файла ще получим пълния stack trace на изключението.

Stack Trace

Информацията, която носи т. нар. **Stack trace**, съдържа подробно описание на естеството на изключението и за мястото в програмата, където то е възникнало. Stack trace се използва, за да се намерят причините за възникването на изключението и последващото им отстраняване (довеждане до нормалното изпълнение на програмата). Stack trace съдържа голямо количество информация и е предназначен за анализиране само от програмистите и администраторите, но не и от крайните потребители на програмата, които не са длъжни да са технически лица. Stack trace е стандартно средство за търсене и отстраняване (дебъгване) на проблеми.

Stack Trace – пример

Ето как изглежда stack trace на изключение за липсващ файл от примера по-горе. Подали сме несъществуващ файл **C:\missingFile.txt** и вместо да изведем съобщението сме използвали метода **e.printStackTrace()**.

```
java.io.FileNotFoundException: C:\missingFile.txt (The system cannot
find the file specified)
   at java.io.FileInputStream.open(Native Method)
   at java.io.FileInputStream.<init>(Unknown Source)
   at java.io.FileInputStream.<init>(Unknown Source)
   at ReadFile.readFile(ReadFile.java:12)
   at ReadFile.main(ReadFile.java:35)
```

Системата не може да намери този файл и затова хвърля изключението **FileNotFoundException**.

Как да разчетем "Stack Trace"?

За да се ориентираме в един stack trace трябва да можем да го разчетем правилно и да знаем неговата структура.

Stack trace съдържа следната информация в себе си:

- пълното име на класа на изключението;
- съобщение – информация за естеството на грешката;
- информация за стека на извикване на методите.

От примера по-горе пълното име на изключението е **java.io.FileNotFoundException**. Следва съобщението за грешка. То донякъде повтаря името на самото изключение: "**C:\missingFile.txt (The system cannot find the file specified)**". Следва целият стек на извикване на методите. Стека най-често е най-голямата част от stack trace.

Всички методи от стека на извикванията са показани на отделен ред. Най-отгоре е методът, който първоначално е хвърлил изключение, а най-отдолу е **main()** методът. Всеки метод се дава заедно с класа, който го съдържа и в скоби реда от файла, където е хвърлено изключението, примерно **ReadFile.readFile(ReadFile.java:12)**. Редовете са налични само ако класът е компилиран с опция да включва дебъг информация (номерата на редовете и т.н.).

Ако методът е конструктор, то вместо името му се използва **<init>** **java.io.FileInputStream.<init>(Unknown Source)**. Ако липсва информация за номера на реда, където е възникнало изключението се изписва **Unknown Source**. Ако методът е **native** (външен за Java виртуалната машина), се изписва **Native Method**.

Това позволява бързо и лесно да се намери класа, метода и дори реда, където е възникнала грешката, да се анализира нейното естество и да се поправи.

Хвърляне на изключения (конструкцията throw)

Изключения се хвърлят с ключовата дума **throw**, като първо се създава инстанция на изключението и се попълва нужната информация за него. Могат да се хвърлят само класове наследници на **java.lang.Throwable**.

Ето един пример:

```
public static void main(String... args) {
   RuntimeException exception = new RuntimeException("Problem");
   throw exception;
}
```

Резултатът от изпълнението на програмата е следният:

```
Exception in thread "main" java.lang.RuntimeException: Problem
   at introjavabook.Program.main(Program.java:10)
```

Видове изключения в Java

В Java има 3 вида изключения: проверени (checked), непроверени (unchecked) и грешки (errors). Когато ги разглеждаме, ще използваме най-вече оригиналните английски термини, защото те са трудно преводими.

В настоящата секция ще се нуждаем от понятия като наследяване, йерархия от класове, базов клас и други, които не сме обяснили до момента. Ако се затруднявате с някой от тях, може да погледнете съответните дефиниции в главата "Принципи на обектно-ориентираното програмиране".

Принципът "хвани или изхвърли"

Принципът "хвани или изхвърли" важи за изключенията, които задължително трябва да се обработят. Те или трябва да бъдат прихванати, или изхвърлени от метода, в който са възникнали, или от някой от следващите методи от стека на извикванията.

За изключенията, които **трябва** да бъдат обработени, има два варианта:

- Изключението да бъде обработено **(хвани)**:

```
public static void openFile(String fileName) {
   try {
      FileInputStream fis = new FileInputStream(fileName);
```

```
    // ...
  } catch (FileNotFoundException e) {
    // ...
  }
}
```

- Отговорността за изключението да бъде оставена на друг **(изхвърли)**, като той бъде задължен да обработи това изключение или да задължи някой друг:

```
public static void openFile(String fileName)
    throws FileNotFoundException {
  FileInputStream fis = new FileInputStream(fileName);
}
```

Изключението или се обработва на място или се обявява като изхвърляно от метода, в който възниква (или хвани или изхвърли). По този начин методът прехвърля отговорността за обработката на даден тип изключения на извикващия метод.

Checked exceptions

Checked (проверени) са изключения, които **задължително трябва** да спазват принципа "хвани или изхвърли" и това се гарантира от компилатора. Тези изключения наследяват класа **java.lang.Exception**, но не наследяват **java.lang.RuntimeException**.

Checked са изключения, които една добре написана програма трябва да очаква и би трябвало да може да се възстанови от тях.

Например програма, която чете данни от сървър с бази от данни. Ако кабелът до сървъра в дадения момент бъде прекъснат, програмата ще получи **ConnectException** и може да съобщи на потребителя да опита отново или да му обясни, че в момента може да използва само други части на програмата.

Checked изключенията или трябва да бъдат прихванати и обработени или трябва да бъдат изхвърляни изрично чрез **throws** декларация в съответния метод. Ако нито едно от двете не е направено, компилаторът ще даде съобщение за грешка.

Грешки (Errors)

Грешките (errors) са критични ситуации (fatal errors), при които изпълнението на програмата обикновено не може да се възстанови и трябва принудително да завърши. Пример за такава грешка е **java.lang.OutOfMemoryError**. Когато паметта свърши, програмата обикновено няма

какво да направи и трябва принудително да запише състоянието си (за да няма загуба на данни) и да завърши.

Грешките не спазват принципа "хвани или изхвърли". Не се очаква да ги обработваме, въпреки че е възможно.

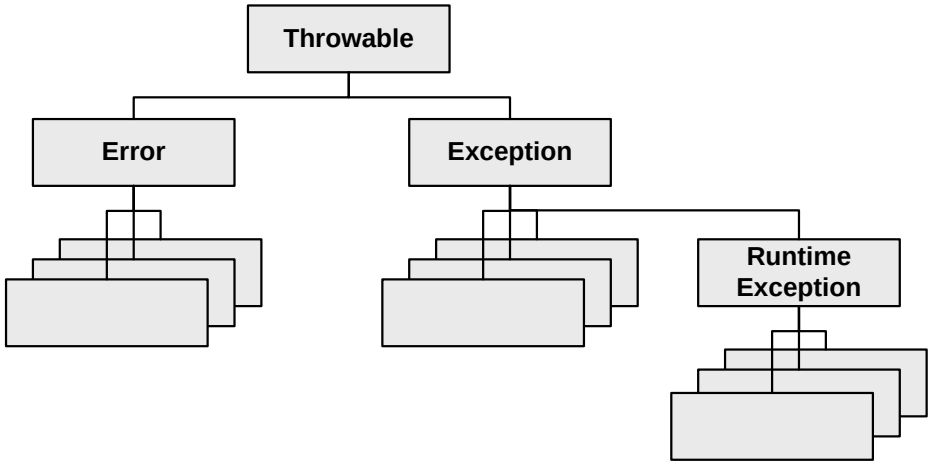
Unchecked exceptions

Unchecked (непроверени) изключения, са изключения, които не са задължени да спазват принципа "хвани или изхвърли". Тези изключения наследяват класа **`RuntimeException`**. Възникването на такова изключение най-често означава бъг в програмата или неправилна употреба на някоя библиотека.

Вероятно сте се сблъсквали с грешката **`NullPointerException`**. Тя е типичен представител на unchecked изключенията. Може да възникне по невнимание, когато се обърнем към обект, който няма стойност. Прихващането и обработването на такива проблеми не е задължително, но е възможно.

Йерархия на изключенията

Изключенията са класове, които образуват йерархия от наследници:



Тъй като наследниците на всеки от тези класове имат различни характеристики, ще разгледаме всеки от тях по отделно.

Throwable

Класът **`java.lang.Throwable`** е базовият клас на всички грешки и изключения в Java. Само този клас или негови наследници могат да се хвърлят от **JVM (Java Virtual Machine – виртуалната машина на Java**) или могат да бъдат хвърляни чрез **`throw`** оператора. Само този клас или негови наследници могат да бъдат аргументи на **`catch`** клаузата.

Throwable – характеристики

Throwable съдържа копие на стека по време на създаването на изключението. Съдържа още текстово съобщение описващо грешката (попълва се от кода, който хвърля изключението или някой наследяващ клас). Всяко изключение може да съдържа още **причина (cause)** за възникването му – друго изключение, което е причина за появата на проблема. Можем да го наричаме **вътрешно / обвито изключение** (**inner / wrapped exception**) или **вложено изключение**.

Външното изключение се нарича **обгръщащо / обвиващо изключение**. Така може да се навържат много изключения. В този случай говорим за **верига от изключения (exception chain)**.

Обвити изключения – защо ги има?

Защо се налага едно изключение да инициира друго?

Добра практика е всеки модул / компонент / програма да дефинира малък брой **application exceptions** (изключения написани от автора на модула / програмата) и този компонент да се ограничава само до тях, а не да хвърля стандартни изключения (от Java API), наричани още **системни изключения (system exceptions)**. Така ползвателят на този модул / компонент знае какви изключения могат да възникнат в него и няма нужда да се занимава с технически подробности.

Например един модул, който се занимава с олихвяването в една банка би трябвало да хвърля изключения само от неговата бизнес област, примерно `InterestCalculationException` и `InvalidPeriodException`, но не и изключения като `FileNotFoundException`, `DivideByZeroException` и `NullPointerException`. При възникване на някое изключение, което не е свързано директно с проблемите на олихвяването, то се обвива в друго изключение от тип `InterestCalculationException` и така извикващия метод получава информация, че олихвяването не е успешно, а като детайли за неуспеха може да разгледа оригиналното изключение, причинител на проблема.

Тези application exceptions от бизнес областта на решавания проблем, за които дадохме пример, обаче не съдържат достатъчно информация за възникналата грешка, за да бъде тя поправена. Затова е добра практика в тях да има и техническа информация за оригиналния причинител на проблема, която е много полезна за дебъгване например.

Същото обяснение от друга гледна точка: един компонент А има дефинирани малък брой изключения (А-изключения). Този компонент използва друг компонент Б. Ако Б хвърли Б-изключение, то А не може да си свърши работата и също трябва да хвърли изключение, но не може да хвърли Б-изключение, затова хвърля А-изключение, съдържащо изключението Б като вложено изключение.

Защо А не може да хвърли Б-изключение? Има много причини:

- Ползвателите на А не трябва да знаят за съществуването на Б (за повече информация разгледайте точката за абстракция от главата за принципите на ООП).
- Компонентът А не е дефинирал, че ще хвърля Б-изключения.
- Ползвателите на А не са подготвени за Б-изключения. Те очакват само А-изключения.

Още една причина ще обясним в секцията за `throws`.

Throwable – методи

Ето най-основните методи на изключенията (класът **`Throwable`**):

```
public class Throwable implements Serializable {
   public Throwable() {
   public Throwable(String message) {
   public Throwable(Throwable cause) {
   public Throwable(String message, Throwable cause) {

   public String getMessage() {...}
   public Throwable getCause() {...}
   public void printStackTrace() {...}
   public StackTraceElement[] getStackTrace() {...}
   public synchronized Throwable initCause(Throwable cause) {
}
```

Нека обясним накратко тези методи:

- Имаме четири конструктора с различните комбинации за съобщение и обвито изключение.
- Методът **`getMessage()`** връща текстово описание на изключението. Ако изключението е **`FileNotFoundException`**, то описанието може да казва кой точно файл не е намерен. Всяко изключение само решава какво съобщение да върне. Най-често се позволява на хвърлящият изключението да сложи това описание.
- Методът **`getCause()`** връща вътрешното / обвитото изключение.
- Методът **`printStackTrace()`** отпечатва класът на изключението, съобщението и стека на грешката и стека на цялата верига от извикани методи, заедно с цялата верига на вложени едно в друго изключения.
- Методът **`getStackTrace()`** връща целия стек, който се пази в изключението. Съществува от Java версия 1.4.
- Методът **`initCause()`** метод използван преди Java 1.4 за задаване на вътрешното / обвитото изключение. Сега се използват конструкторите, но този метод също продължава да работи.

Вериги изключения – частта Caused by

Сега ще видим как се изписва на екрана вложено изключение. Нека имаме следния код:

```
package introjavabook;

public class ExceptionProgram {

   public static void main(String... args) {
      try {
         throw new NullPointerException("Problem");
      } catch (NullPointerException npe) {
         throw new RuntimeException(npe);
      }
   }
}
```

В този пример хвърляме едно изключение, след това го хващаме и хвърляме друго изключение, което обвива първото. Резултатът от изпълнението този код е:

```
Exception in thread "main" RuntimeEx: NullPointerEx: Problem
   at ExceptionProgram.main(ExceptionProgram.java:13)
Caused by: java.lang.NullPointerException: Problem
   at ExceptionProgram.main(ExceptionProgram.java:11)
```

На места са съкратени имената, за да се събират на един ред. Появява се секция "Caused by", която описва вложеното изключение. Това ни дава полезна информация за това как се е стигнало до хвърлянето на изключението, което разглеждаме.

Как да разчетем "съкратен" Stack Trace?

Понякога получаваме изключение като това:

```
HighLevelException: MidLevelException: LowLevelException
    at Junk.a(Junk.java:14)
    at Junk.main(Junk.java:4)
Caused by: MidLevelException: LowLevelException
    at Junk.c(Junk.java:24)
    at Junk.b(Junk.java:18)
    at Junk.a(Junk.java:12)
    ... 1 more
Caused by: LowLevelException
    at Junk.e(Junk.java:31)
    at Junk.d(Junk.java:28)
    at Junk.c(Junk.java:22)
```

```
    ... 3 more
```

Какво означава частта "**... 3 more**"? Тя означава, че за краткост при отпечатването на грешката някои подробности за нея са били съкратени, тъй като са очевидни. В случая се пропускат повтарящите се редове. Ето пълната версия на същия "stack trace":

```
HighLevelException: MidLevelException: LowLevelException
    at Junk.a(Junk.java:14)
    at Junk.main(Junk.java:4)
Caused by: MidLevelException: LowLevelException
    at Junk.c(Junk.java:24)
    at Junk.b(Junk.java:18)
    at Junk.a(Junk.java:12)
    ... 1 more = (последният един ред от горния стек)
=> at Junk.main(Junk.java:4)

Caused by: LowLevelException
    at Junk.e(Junk.java:31)
    at Junk.d(Junk.java:28)
    at Junk.c(Junk.java:22)
    ... 3 more = (последните три реда от горния стек)
=>  at Junk.b(Junk.java:18)
    at Junk.a(Junk.java:12)
    at Junk.main(Junk.java:4) (взето от най-горния)
```

Съкращаването на повтарящи се редове "компресира" изключенията – прави ги по-кратки. Това обикновено е полезно и затова се извършва автоматично при отпечатване.

Error

Изключенията от тип **Error** и неговите наследници индикират за сериозен проблем (неочаквани грешки). Използват се при възникване на грешки – такива, от които програмата не може да се възстанови и не се очаква те да бъдат хващани. Всички такива грешки наследяват класа **Error**.

Тъй като класа Error наследява Throwable, той притежава всички негови свойства: носи в себе си stack trace и съобщение за грешка, съдържа методите **getMessage()**, **printStackTrace()** и **getStackTrace()** и може да съдържа вложено изключение, достъпно с **getCause()**.

Пример за такова изключение е **OutOfMemoryError**, което възниква при заделяне на памет, когато не може да бъде отделено достатъчно място за новосъздаващия се обект.

Exception

Това са т. нар checked изключения. Те наследяват класа **Exception**, който от своя страна наследява **Throwable**. По този начин всички checked изключения имат stack trace, могат да съдържат други изключения и въобще имат всички методи и свойства на **Throwable**.

Такива изключения или трябва да се прихващат или да се изхвърлят (хвани или изхвърли).

Пример за такова изключение е **IOException**, което възниква при проблем по време на работа с входно-изходни операции. Това изключение задължително трябва да се хване или да се изхвърли.

RuntimeException

Това са т. нар unchecked изключения. Те наследяват **RuntimeException**, който от своя страна наследява **Exception**, а той наследява **Throwable**. Следователно unchecked изключенията също имат всички свойства и методи от класа **Throwable** (stack trace, message, cause и др.). Прихващането на тези изключения не е задължително.

Декларацията throws за методи

Досега разгледахме програмни конструкции за прихващане на изключения в рамките на даден метод, но понякога е подходящо изключенията да бъдат изхвърлени от метода и отговорност за тяхната обработка да поеме извикващият метод. За да се постигне това, не трябва да прихващаме изключенията, а просто да декларираме те да се пропуснат към предишния метод в стека. Това става посредством добавка в сигнатурата на метода. Изброяваме всички изключения, които искаме да се пропуснат в списък с разделител запетая, започващ с ключовата дума **throws**. Списъкът се намира в края на сигнатурата на метода след списъка с параметрите и преди отварящата фигурна скоба:

```
public static void readFile(String fileName)
     throws FileNotFoundException, IOException {
}
```

Това е примерен метод, които изхвърля два типа изключения - **FileNotFoundException** и **IOException**. В случая **FileNotFoundException** е излишно обявен, тъй като е наследник на **IOException**. Понякога този запис се предпочита заради по-добрата четимост на кода. За по-добра четимост на кода понякога се декларират за изхвърляне от метода дори **Runtime** изключения, въпреки че това не се изисква от компилатора.

Да се върнем на първоначалния ни пример. Единственото, което прибавяме, за да се компилира методът, е декларацията **throws FileNotFoundException, IOException**.

```
public static void readFile(String fileName)
      throws FileNotFoundException, IOException {
   FileInputStream fis = null;
   fis = new FileInputStream(fileName);
   BufferedReader in = new BufferedReader(
      new InputStreamReader(fis));
   String tmp = null;
   while ((tmp = in.readLine()) != null) {
      System.out.println(tmp);
   }
   in.close();
   fis.close();
}
```

Изхвърлянето на изключения към предходния метод обикновено означава, че изключенията трябва да се прихванат и обработят там. Най-правилното място да се обработи дадено изключение е там, където се изпълнява бизнес операцията, която е била прекъсната поради някакъв проблем. Това е сложно за обяснение, но по принцип не трябва да прихващаме изключения, които не можем да обработим адекватно. В такива случаи е по-добре да ги декларираме с **throws** и да не се занимаваме с тях.

Във визуалните (GUI) приложения грешката трябва да се покаже на потребителя под формата на диалогов прозорец съдържащ описание, съобразено с познанията на потребителите. В конзолните приложения обикновено грешката се изписва на конзолата. При уеб приложения грешката се визуализира като червен текст в началото на страницата или след полето, за което се отнася.

Има едно универсално правило за обработката на изключенията:

Един метод трябва да обработва само изключенията, за които е компетентен, които очаква и за които има знания как да ги обработи. Останалите трябва да ги изхвърля към извикващия метод.

Ако изключенията се предават по гореописания начин от метод на метод и не се прихванат никъде, те неминуемо ще достигнат до началния метод от програмата – **main()** метода – и ако и той не ги прихване, виртуалната машина ще ги отпечата на конзолата (ако има конзола) и ще преустанови изпълнението на програмата.

Обвити изключения – защо ги има?

Нека компонентът А хвърля checked изключения – А-изключения. Той използва функционалност на компонента Б. Компонентът Б хвърля Б-изключения, които също са checked. Тогава, ако в метод на Б възникне Б-изключение, то методът на А, който ползва този метод, не може да хвърли Б-изключение. Това, което може да направи, е или да обработи изключението, или да го обвие в **RuntimeException** или да го обвие в А-изключение. Най-препоръчителна е практиката Б-изключението да бъде обвито в А-изключение. Това е още една причина, поради която съществуват обвитите изключения (nested exceptions).

Изхвърляне на изключения от main() метода – пример

Изхвърлянето на изключения от **main()** метода по принцип не е желателно и до версия 1.4 на JDK не е позволено. Желателно е всички изключения да бъдат прихванати и обработени. От версия Java 5 изхвърлянето на изключения от **main()** метода е възможно, както от всеки друг метод:

```
public static void main(String a[])
     throws FileNotFoundException, IOException {
   readFile("C:\\tools\\eclipse\\eclipse.ini");
}
```

Всички изключения изхвърлени от **main()** метода се прихващат от самата виртуална машина и се обработват по един и същ начин – пълният stack trace на изключението се изписва в стандартния изход за грешки (**System.err**). Такова изхвърляне на изключенията, възникващи в **main()** метода е много удобно, когато пишем кратка програмка набързо и не искаме да обработваме евентуално възникващите изключения.

Прихващане на изключения на нива – пример

Възможността за пропускане на изключения през даден метод ни позволява да разгледаме един по-сложен пример. Прихващане на изключения на нива. Прихващането на нива е комбинация от прихващането на определени изключения в едни методи и пропускане на други изключения към предходните методи (нива) в стека. В примера изключенията възникващи в метода **readFile()** се прихващат на две нива:

```
public static void main(String[] args) {
   try {
      readFile("C:\\tools\\eclipse\\eclipse.ini");
   } catch (IOException e) {
      e.printStackTrace();
```

```
   }
}

public static void readFile(String fileName) throws IOException{
   try {
      FileInputStreamfis = new FileInputStream(fileName);
      BufferedReader in = new BufferedReader(
         new InputStreamReader(fis));
      String tmp = null;
      while ((tmp = in.readLine()) != null) {
         System.out.println(tmp);
      }
   } catch (FileNotFoundException e) {
      System.out.println("The file \"" + fileName +
         "\" does not exist! Unable to read it.");
   }
}
```

Първото ниво на прихващане е в метода **readFile()** а второто ниво е в **main()** метода. Методът **readFile()** прихваща само изключенията от тип **FileNotFoundException**, а пропуска всички останали **IOException** изключения към **main()** метода където те биват прихванати и обработени.

Ако **main()** метода подаде име на несъществуващ файл то ще възникне **FileNotFoundException** което ще се прихване в **readFile()**. Ако обаче се подаде име на съществуващ файл и възникне грешка при самото четене на файла то изключението ще се прихване в **main()** метода.

Прихващането на изключения на нива позволява отделните изключения да се обработват на най-подходящото място.

Конструкцията try-finally

Всеки блок **try** може да съдържа блок **finally**. Блока **finally** се изпълнява **винаги** при излизане от **try** блока, независимо как се излиза от **try** блока. Това гарантира изпълнението на **finally** блока дори ако възникне неочаквано изключение или се излезе с израз **return**.

Блокът finally няма да се изпълни, ако по време на изпълнението на блока try виртуалната машина прекрати изпълнението си!

Блокът **finally** има следната основна форма:

```
try {
   Some code that could or could not cause an exception
} finally {
```

```
   // Code here will allways execute
}
```

Всеки **try** блок може да има един единствен блок **finally** освен блоковете **catch**. Възможна е и комбинация с множество **catch** блокове и един **finally** блок.

```
try {
   some code
} catch (...) {
   // Code handling an exception
} catch (...) {
   // Code handling another exception
} finally {
   // This code will allways execute
}
```

В случай на нужда от освобождаване на вече заети ресурси блока **finally** е незаменим. Ако го нямаше, никога не бихме били сигурни дали разчистването няма случайно да се прескочи при неочаквано изключение или заради използването на **return**, **continue**, или **break** изрази.

Дефиниране на проблема

В примера, който разглеждаме има два потока, които задължително трябва да се затворят. Най-правилният начин това да се направи е с **try- finally** блок обграждащ редовете, където се използват съответните потоци. Да си припомним примера:

```
public static void readFile(String fileName) throws IOException{
   FileInputStream fis = new FileInputStream(fileName);
   BufferedReader in =
      new BufferedReader(new InputStreamReader(fis));
   // Using the streams here …
   in.close();
   fis.close();
}
```

Отварят се два потока – един **FileInputStream** и един **BufferedReader**. Следва използването на потоците накрая следва задължителното им затваряне преди да се излезе от метода. Задължителното затваряне на потоците е проблемна ситуация, защото от метода може да се излезе по няколко начина:

- По време на инициализиране на първия поток възникне непредвидено изключение.

- По време на инициализиране на втория поток възникне непредвидено изключение.
- По време на използването на потоците възниква непредвидено изключение.
- Между инициализирането и затварянето на потоците се използват израза **return**.
- Всичко е нормално и не възникват никакви изключения.

Така написан примерът е логически грешен, защото потоците ще се затворят правилно само в първия и последния случай. Във втория случай вторият поток няма да се затвори, а в третия и четвъртия случай и двата потока няма да се затворят. Тук не взимаме под внимание възможността отварянето използването и затварянето на потоците да е част от тяло на цикъл, където може да се използва изразите **continue** и **break**, което също ще доведе до не затваряне на потоците.

Решение на проблема

Всички тези главоболия можем да си спестим като използваме конструкцията **try-finally**. Ще разгледаме първо пример с един поток, а след това и за два и повече потока.

Сигурното затваряне на поток се прави по следния начин:

```
FileInputStream fis = null;
try {
   fis = new FileInputStream("fileName.txt");
   // Using "fis" here ...
} finally {
   // Always close "fis"
   if (fis != null) {
      fis.close();
   }
}
```

Да анализираме примера. Първоначално декларираме променлива **fis** от тип **FileInputStream**, след това отваряме **try** блок, в който инициализираме нов файлов поток, използваме го и накрая го затваряме във **finally** блок. Каквото и да стане при използването и инициализацията сме сигурни, че потока ще бъде затворен. Ако има проблем при инициализацията – например липсващ файл то ще се хвърли **FileNotFoundException** и променливата **fis** ще остане със стойност **null**. За тези случай и за да се избегне **NullPointerException** е необходимо да се прибави проверка дали **fis** не е **null** преди да се извика метода **close()** на потока. Ако имаме **null** то потока изобщо не е бил инициализиран и няма нужда да бъде затварян.

Горния пример трябва подходящо да обработи всички checked exceptions, които възникват при инициализиране (**FileNotFoundException**) и използване на файловия поток. В примера възможните изключения просто се изхвърлят от метода. Понеже те всички са наследници на **IOException** в декларацията на метода се използва само това изключение.

Даденият пример е за файлови потоци, но може да се използва за произволни ресурси, които изискват задължително освобождаване след приключване на работата с тях. Такива ресурси могат да бъдат връзки към отдалечени компютри, връзки с бази данни и др.

Алтернативно решение

Опростена версия на горния пример се явява следната конструкция:

```
FileInputStream fis = new FileInputStream("fileName.txt");
try {
   // Using "fis" here …
} finally {
   fis.close();
}
```

Предимството е по-краткия запис – спестяваме една излишна декларация на променливата **fis** и избягваме проверката за **null**. Проверката за **null** е излишна, защото инициализацията на потока е извън **try** блока и ако е възникнало изключение докато тя се изпълнява изобщо няма да се стигне до изпълнение на **finally** блока и затварянето на потока.

Недостатък е невъзможността да се обработят изключения възникнали при инициализацията в същия **try** блок. Трябва да използваме допълнителен **try** блок, който да прихване всички възможни checked exceptions по време на инициализацията или да ги изхвърлим от самия метод.

Освобождаване на множество ресурси

Досега разгледахме използването на **try-finally** за освобождаване на един ресурс, а в примера имаме да затворим два потока. Добра практика е ресурсите да се освобождават в ред обратен на този на заделянето им – в нашия пример първо заделяме файлов, а след това и буфериран поток. Трябва да ги освободим в обратния ред – първо буферирания, после и файловия поток.

За освобождаването на множество ресурси могат да се използват горните два подхода като **try-finally** блоковете се влагат един в друг:

```
FileInputStream fis = new FileInputStream(fileName);
try {
   BufferedReader in = new BufferedReader(
```

```
      new InputStreamReader(fis));
   try {
      // Using "in" here
   } finally {
      in.close();
   }
} finally {
   fis.close();
}
```

Другият вариант е всички ресурси да се декларират предварително и накрая да се освободят в един единствен **finally** блок с проверка за **null**:

```
FileInputStream fis = null;
BufferedReader in = null;
try {
   fis = new FileInputStream(fileName);
   in = new BufferedReader(new InputStreamReader(fis));
   // Using "in" here …
} finally {
   if (in != null) {
      in.close();
   }
   if (fis != null) {
      fis.close();
   }
}
```

И двата подхода са правилни със съответните предимства и недостатъци и се прилагат в зависимост от предпочитанията на програмиста съобразно конкретната ситуация. Все пак вторият подход е малко рисков, тъй като във **finally** блока възникне изключение (което почти никога не се случва) при затварянето на първия поток, вторият поток няма да бъде затворен. При първия подход няма такъв проблем, но се пише повече код.

Време е да се върнем на нашия примерен метод и да го напишем по правилен начин. Да използваме влагането на **try-finally** блокове описано преди малко.

```
public static void readFile(String fileName)
      throws IOException {
   FileInputStream fis = new FileInputStream(fileName);
   try {
      BufferedReader in = new BufferedReader(
         new InputStreamReader(fis));
      try {
        // Using "in" here …
```

```
      } finally {
         in.close();
      }
   } finally {
      fis.close();
   }
}
```

А сега, нека обобщим всичко научено досега в един общ пример.

Обобщение

Примерният клас **ReadFile** обобщава наученото досега и групира в една цялостна програма примерите разглеждани до момента:

ReadFile.java

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReadFile {
   public static void readFile(String fileName)
         throws IOException {
      FileInputStream fis = null;
      BufferedReader in = null;
      try {
         fis = new FileInputStream(fileName);
         in = new BufferedReader(new InputStreamReader(fis));
         String tmp = null;
         while ((tmp = in.readLine()) != null) {
            System.out.println(tmp);
         }
      } catch (FileNotFoundException e) {
         System.out.println("The file \"" + fileName
                     + "\" does not exist! Unable to read it.");
      } finally {
         if (in != null) {
            in.close();
         }
         if (fis != null) {
            fis.close();
         }
      }
   }
```

```
   public static void main(String[] args) {
      try {
         readFile("C:\\tools\\eclipse\\eclipse.ini");
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Голяма част от възможностите на изключенията са илюстрирани с този пример. Това са прихващане на изключение, прихващане на нива, прихващане на група от изключения, пропускане на изключения към извикващ метод в стека, обработка на изключения и почистване на заети ресурси.

Спорът около checked изключенията

Няколко библиотеки в Java използват checked изключения доста обширно. Библиотеките за работа с потоци, за комуникация по мрежата, за работа с файлове, JDBC (за работа с релационни бази от данни) – всички те използват главно checked изключения.

Има твърдения, че в тези авторите на тези библиотеки са прекалили с употребата на такъв вид изключения и че това трябва да се промени. Това е един обширен спор, който се води в Java общността. Защитниците на тази теза дават следния пример:

```
package introjavabook;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class CatchInFinally {
   public static void main(String... args) {
      FileInputStream fis = null;
      try {
         fis = new FileInputStream("file.bin");
         byte[] data = new byte[10];
         fis.read(data, 0, 10);
      } catch(FileNotFoundException e) {
         e.printStackTrace();
      } catch(IOException e) {
         e.printStackTrace();
      } finally { // Note: we have try-catch in the finally block
         try {
            fis.close();
```

```
        } catch(IOException e) {
           e.printStackTrace();
        }
     }
   }
}
```

Тези защитници твърдят, че в повечето случаи тези изключения не са толкова важни, за да бъдат checked – което предполага обработка.

От Sun, обаче са против тази промяна и предпочитат разработчиците да пишат повече код, но да са сигурни, че са предвидили всяка една ситуация.

В общността има предложения **close()** методът във всички потоци да се промени, така че да не хвърля checked exception (както е в .NET Framework), но дали това изменение ще бъде въведено в някоя следваща версия на Java платформата можем само да гадаем.

Предимства при използване на изключения

След като се запознахме подробно с изключенията, техните свойства и с това как да работим с тях, нека разгледаме причините те да бъдат въведени и да придобият широко разпространение.

Отделяне на кода за обработка на грешките

Използването на изключения позволява да се отдели кода описващ нормалното протичане на една програма от кода необходим в изключителни ситуации и кода необходим при обработване на грешки. Това ще демонстрираме със следния пример, които е псевдокод на примера разгледан от началото на главата.

```
readFile() {
   openFileInputStream();
   while (fileHasMoreLines) {
      readNextLine();
      printTheLine();
   }
   closeTheFile();
}
```

Нека сега преведем последователността от действия на български:

- Отваряме файл;
- Четем следващ ред от файла до края;
- Изписваме прочетения ред;
- Затваряме файла;

Методът е добре написан, но ако се вгледаме по-внимателно започват да възникват въпроси:

- Какво ще стане, ако няма такъв файл?
- Какво ще стане, ако файлът не може да се отвори (например, ако друг процес вече го е отворил за писане)?
- Какво ще стане, ако пропадне четенето на някой ред?
- Какво ще стане, ако файла не може да се затвори?

Да допишем метода, така че да взима под внимание тези въпроси, без да използваме изключения, а да използваме кодове за грешка връщани от всеки използван метод (кодовете за грешка са стандартен похват за обработка на грешките в процедурно ориентираното програмиране. Всеки метод връща **int**, който определя дали методът е изпълнен правилно. Код за грешка 0 означава, че всичко е правилно, код различен от 0 означава различен тип грешка).

```
errorCode readFile() {
   errorCode = 0;
   openFileErrorCode = openFileInputStream();

   // File is open
   if (openFileErrorCode = 0) {
      while (fileHasMoreLines) {
         readLineErrorCode = readNextLine();
         if (readLineErrorCode == 0) {
            // Line read properly
            printTheLine();
         } else {
            // Error during line read
            errorCode = -1;
            break;
         }
      }
      closeFileErrorCode = closeTheFile();
      if (closeFileErrorCode != 0 && errorCode == 0) {
         errorCode = -2;
      } else {
         errorCode = -3;
      }
   } else if (openFileErrorCode = -1) {
      // File does not exists
      errorCode = -4;
   } else if (openFileErrorCode = -2) {
      // File can't be open
      errorCode = -5;
   }
```

```
   return errorCode;
}
```

Получава се един доста замотан, трудно разбираем и лесно объркващ – "спагети" код. Логиката на програмата е силно смесена с логиката за обработка на грешките и непредвидените ситуации. По-голяма част от кода е тази за правилна обработка на грешките.

Всички тези нежелателни последици се избягват при използването на изключения. Ето колко по-прост и чист е псевдокода на същия метод, само че с изключения:

```
readFile() {
   try {
      openFileInputStream();
      while (fileHasMoreLines) {
         readNextLine();
         printTheLine();
      }
      closeTheFile();
   } catch (FileNotFoundException) {
      doSomething;
   } catch (IOException) {
      doSomething;
   }
}
```

Всъщност изключенията не ни спестяват усилията при намиране и обработка на грешките, но ни позволяват да правим това по далеч по-елегантен, и ефективен начин.

Групиране на различните видове грешки

Йерархичната същност на изключенията позволява наведнъж да се прихванат и обработят цели групи от тях. Когато използваме **catch**, ние не прихващаме само дадения тип изключение, а цялата йерархия на типовете изключения, наследници на декларирания от нас тип.

```
} catch (IOException e) {
   // Handle IOException and all its descendants
}
```

Горният пример ще прихване не само IOException, но и всички негови наследници в това число **FileNotFoundException**, **EOFException**, **RemoteException** и десетки други.

Въпреки че не е добра практика, е възможно да направим прихващане на абсолютно всички изключения:

```
} catch (Exception e) {
   // A (too) general exception handler
}
```

И дори на всичко, което може да се хвърля – всички **Throwable** класове.

```
} catch (Throwable e) {
   // A (too, too) general exception handler
}
```

Прихващането на **Exception** и **Throwable** не е добра практика и се предпочита прихващането на по-конкретни групи от изключения като **IOException** или на един единствен тип изключение като например **FileNotFoundException**.

Предаване на грешките за обработка в стека на методите – прихващане на нива

Възможността за прихващането на изключения на нива е изключително удобна. Тя позволява обработката на изключението да се направи на най-подходящото място. Нека илюстрираме това с прост пример-сравнение с остарелите кодове за грешка. Нека имаме следната структура от методи:

```
method3() {
   method2();
}

method2() {
   method1();
}

method1() {
   readFile();
}
```

Метода **method3()** извиква **method2()**, който от своя страна извиква **method1()** където се вика **readFile()**. Да предположим, че **method3()** е този който се интересува от възможна възникнала грешка в метода **readFile()**. Нека сега си представим, че възникне грешка в метода **readFile(),** която трябва да се обработи от **method3()**. Това не би било никак лесно:

```
method3() {
   errorCode = method2();
   if (errorCode != 0)
      process the error;
   else
```

```
      do actual work;
}

errorCode method2() {
   errorCode = method1();
   if (errorCode != 0)
      return errorCode;
   else
      do actual work;
}

errorCode method1() {
   errorCode = readFile();
   if (errorCode != 0)
      return errorCode;
   else
      do actual work;
}
```

Като начало в **method1()** трябва анализираме кода за грешка връщан от метода **readFile()** и евентуално да предадем на **method2()**. В **method2()** трябва да анализираме кода за грешка връщан от **method1()** и евентуално да го предадем на **method3()**, където да се обработи самата грешка.

Как можем да избегнем всичко това? Да си припомним, че виртуалната машина търси прихващане на изключения назад в стека на извикване на методите и позволява на всеки един от методите в стека да дефинира прихващане и обработка на изключението. Ако методът не е заинтересован, чрез **throws** клаузата, просто препраща изключението по-назад в стека:

```
method3() {
   try {
      method2();
   } catch (exception e) {
      process the exception;
   }
}

method2() throws exception {
   method1();
}

method1() throws exception {
   readFile();
}
```

Ако възникне грешка при четенето на файл, то тя ще се пропусне от **method1()** и **method2()** и ще се прихване и обработи чак в **method3(),** където всъщност е подходящото място за обработка на грешката.

Както се вижда от псевдокода методите **method2()** и **method1()** все пак трябва да знае какво да пропусне чрез клаузата **throws**.

Добри практики при работа с изключения

В настоящата секция ще дадем някои препоръки и утвърдени практики за правилно използване на механизмите на изключенията за обработка на грешки и необичайни ситуации. Това са важни правила, които трябва да запомните и следвате. Не ги пренебрегвайте!

Кога да разчитаме на изключения?

За да разберем кога е добре да разчитаме на изключения и кога не, нека разгледаме следния пример:

Имаме програма, която отваря файл по зададени път и име на файл от потребителя. Потребителят може да объркa името на файла докато го пише. Тогава това събитие по-скоро трябва да се счита за нормално, а не за изключително.

Срещу подобно събитие можем да се защитим като първо проверим дали файлът съществува и чак тогава да се опитаме да го отворим:

OpenFileTest.java

```
package introjavabook;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class OpenFileTest {

   public static void main(String... args) {
      File f = new File("f.txt");
      if (!f.exists()) {
         System.out.println("The file does not exists.");
         return;
      }

      Scanner scan = new Scanner(f);
      String line = scan.nextLine();
      System.out.println("The first line of the file: " + line);
   }
}
```

Ако файлът липсва, ще получим съобщението:

```
The file does not exists.
```

Другият вариант да имплементираме същата логика е следният:

```
File f = new File("f.txt");
Scanner scan;
try {
   scan = new Scanner(f);
} catch (FileNotFoundException e) {
   System.out.println("The file does not exists.");
   return;
}

String line = scan.nextLine();
System.out.println("The first line of the file: " + line);
```

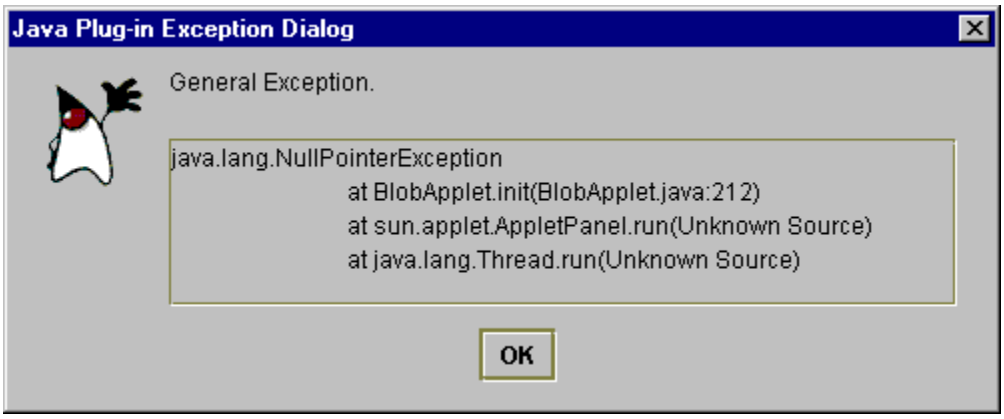
По принцип вторият вариант се счита за по-лош, тъй като изключенията трябва да се ползват за изключителна ситуация, а липсата на файла в нашия случай е по-скоро обичайна ситуация.

Лоша практика е да се разчита на изключения за обработка на очаквани събития и от гледна точка на производителността. Хвърлянето на изключение е бавна операция – трябва да се инициализира stack trace, трябва да се открие обработчик на това изключение и т.н.

Точната граница между очаквано и неочаквано поведение е трудно да бъде ясно дефинирана. Най-общо очаквано събитие е нещо свързано с функционалността на програмата. Въвеждането на грешно име на файла е пример за такова. Спирането на тока докато работи тази програма, обаче не е очаквано събитие.

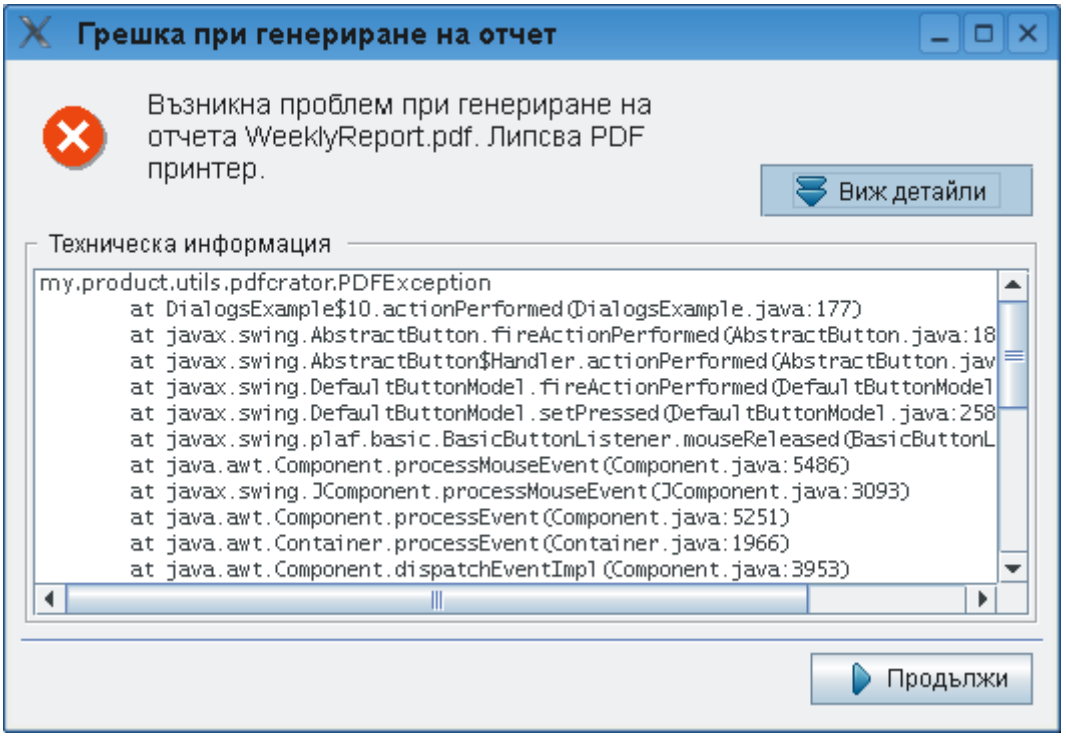
Да хвърляме ли изключения на потребителя?

Изключенията са неясни и объркващи за обикновения потребител. Те създават впечатление за лошо написана програма, която "гърми неконтролирано" и има бъгове. Представете си какво ще си помисли една възрастна лелка, което въвежда фактури, ако внезапно приложението ѝ покаже следния диалог:



Този диалог е много подходящ за технически лица, но е изключително неподходящ за крайния потребител (особено, когато той няма технически познания).

Вместо този диалог можем да покажем друг, много по-дружелюбен и разбираем за обикновения потребител:



Това е добрият начин да показваме съобщения за грешка: хем да има разбираемо съобщение на езика на потребителя (в случая на български език), хем да има и техническа информация, която може да бъде извлечена при нужда, но не се показва в самото начало, за да не стряска потребителите.

Препоръчително е изключения, които не са хванати от никой, а такива може да са само runtime изключенията или грешките, да се хващат от общ глобален "прихващач", който да ги записва (в най-общия случай) някъде, а на потребителя да показва "приятелско" съобщение в стил: "Възникна грешка, опитайте по-късно". Добре е винаги да показвате освен съобщение разбираемо за потребителя и техническа информация (stack trace), която, обаче е достъпна само ако потребителят я поиска.

Хвърляйте изключенията на съответното ниво на абстракция!

Когато хвърляте ваши изключения, съобразявайте се с абстракциите, в контекста, на които работи вашият метод. Например, ако вашият метод се отнася за работа с масиви, може да хвърлите и **`ArrayIndexOutOfRange`** или **`NullPointerException`**, тъй като вашият метод работи на ниско ниво и оперира директно с паметта и с елементите на масивите. Ако, обаче имате метод, който извършва олихвяване на всички сметки в една банка, той не трябва да хвърля **`ArrayIndexOutOfRange`**, тъй като това изключение не е от бизнес областта на банковия сектор и олихвяването. Нормално е олихвяването да хвърли изключение **`InvalidInterestException`** с подходящо съобщение за грешка от бизнес областта на банките, за което би могло да бъде закачено оригиналното изключение **`ArrayIndexOutOfRange`**.

Представете си да сте си купили билет за автобус и пристигайки на автогарата омаслен монтьор да ви обясни, че ходовата част на автобуса има нужда от реглаж. Освен, ако не сте монтьор или специалист по автомобили, тази информация не ви помага с нищо. Нито става ясно колко ще се забави вашето пътуване, нито дали въобще ще пътувате. Вие очаквате, ако има проблем да ви посрещне усмихната девойка от фирмата-превозвач и да ви обясни, че резервният автобус ще дойде след 10 минути и до тогава можете да изчакате на топло в кафенето.

Същото е с програмирането – ако хвърляте изключения, които не са от бизнес областта на компонента или класа, който разработвате, има голям шанс да не ви разберат и грешката да не бъде обработена правилно.

Можем да дадем още един пример: извикваме метод, който сортира масив с числа и той хвърля изключение **`DatabaseTransactionAborted`**. Това е също толкова неадекватно съобщение, колкото и **`NullPointerException`** при изпълнение на олихвяването в една банка. Веднага ще си помислите "каква транзакция, нали сортираме масив" и този въпрос е напълно адекватен. Затова се съобразявайте с нивото на абстракция, на което работи дадения метод, когато хвърляте изключение от него.

Ако изключението има причинител, запазвайте го!

Винаги, когато при прихващане на изключение хвърляте ново изключение от по-високо ниво на абстракция, добавяйте към него оригиналното

изключение. По този начин ползвателите на вашия код ще могат по-лесно да установят точната причина за грешката и точното място, където тя възниква в началния момент. Това правила е частен случай на по-генералното правило, че всяко изключение трябва да носи в себе си максимално подробна информация за настъпилия проблем.

Давайте подробно описателно съобщение при хвърляне на изключение!

Съобщението за грешка, което всяко изключение носи в себе си е изключително важно. В повечето случаи то е напълно достатъчно, за да разберете какъв точно е проблемът, който е възникнал. Ако съобщението е неадекватно, ползвателите на вашия метод няма да са щастливи и няма да решат бързо проблема.

Да вземем един пример: имате метод, който прочита настройките на дадено приложение от текстов файл. Това са примерно местоположенията и размерите на всички прозорци в приложението и други настройки. Случва се проблем при четенето на файла с настройките и получавате съобщение за грешка:

```
Error.
```

Това достатъчно ли ви е, за да разберете какъв е проблемът? Очевидно не е. Какво съобщение трябва да дадем, така че то да е достатъчно информативно? Това съобщение по-добро ли е?

```
Error reading settings file.
```

Очевидно горното съобщение е по-адекватно, но е недостатъчно. То обяснява каква е грешката, но не обяснява причината за възникването ѝ. Да предположим, че променим програмата, така че да дава следната информация за грешката:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings
```

Това съобщение очевидно е по-добро, защото ни подсказва в кой файл е проблемът (нещо, което би ни спестил много време, особено ако не сме запознати с приложението и не знаем къде точно то пази файла с настройките си). Може ситуацията да е дори по-лоша – може да нямаме сорс кода на въпросното приложение или модул, който генерира грешката. Тогава е възможно да нямаме пълен stack trace (ако сме компилирали без дебъг информация) или ако имаме stack trace, той не ни върши работа, защото нямаме сорс кода на проблемния файл, хвърлил изключението. Затова съобщението за грешка трябва да е още по-подробно, например като това:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings. Number expected at line
17.
```

Това съобщение вече само говори за проблема. Очевидно имаме грешка на ред 17 във файла **MyApp.settings**, който се намира в папката **C:\Users\Administrator\MyApp**. В този ред трябва да има число, а има нещо друго. Ако отворим файл, бързо можем да намерим проблема, нали?

Изводът от този пример е само един:

Винаги давайте адекватно, подробно и конкретно съобщение за грешка, когато хвърляте изключение! Ползвателят на вашия код трябва само като прочете съобщението, веднага да му стане ясно какъв точно е проблемът, къде се е случил и каква е причината за него.

Ще дадем още няколко примера:

- Имаме метод, който търси число в масив. Ако той хвърли **IndexOutOfRangeException**, от изключително значение е индексът, който не може да бъде достъпен, примерно 18 при масив с дължина 7. Ако не знаем позицията, трудно ще разберем защо се получава излизане от масива.
- Имаме метод, който чете числа от файл. Ако във файла се срещне някой ред, на който няма число, би трябвало да получим грешка, която обяснява, че на ред 17 (примерно) се очаква число, а там има стринг (и да се отпечата точно какъв стринг има там).
- Имаме метод, който изчислява стойността на числен израз. Ако намерим грешка в израза, изключението трябва да съобщава каква грешка е възникнала и на коя позиция. Кодът, който предизвиква грешката може да ползва **String.format**, за да изгради съобщението за грешка. Ето един пример:

```
throw new ExpressionParseException(
   String.format("Invalid character at position %d." +
      "Number expected but found character '%s'.", index, ch);
```

Има само едно нещо по-лошо от изключение без достатъчно информация и то е изключение с грешна информация. Например, ако в последния пример съобщим за грешка на ред 3, а грешката е на ред 17, това е изключително заблуждаващо и е по-вредно, отколкото просто да кажем, че има грешка без подробности. Внимавайте да не отпечатвате грешни съобщения за грешка!

За съобщенията за грешки използвайте английски език

Това правило е много просто. То е частен случай на принципа, че целият сорс код на програмите ви (включително коментарите и съобщенията за грешки) трябва да са на английски език. Причината за това е, че това е единственият език, който е разбираем за всички програмисти по света. Никога не знаете дали кодът, който пишете няма в някой слънчев ден да се ползва от чужденци. Хубаво ли ще ви е, ако ползвате чужд код и той ви съобщава за грешки примерно на виетнамски език?

Никога не игнорирайте прихванатите изключения!

Никога не игнорирайте изключенията, които прихващате, без да ги обработите. Ето един пример как не трябва да правите:

```
FileInputStream fis = null;
try {
   fis = new FileInputStream("file.bin");
   byte[] data = new byte[10];
   fis.read(data, 0, 10);
} catch(FileNotFoundException e) {
} catch(IOException e) {
} finally {
   try {
      fis.close();
   } catch(IOException e) {
   }
}
```

В този пример авторът на този ужасен код прихваща изключенията и ги игнорира. Това означава, че ако липсва файлът, който търсим, програмата няма да прочете нищо от него, но няма и да съобщи за грешка. Ползвателят на този код бъде заблуден, че файлът е бил прочетен, а той всъщност липсва.

Начинаещите програмисти понякога пишат такъв код, защото компилаторът ги задължава да прихващат изключенията при работа с файлове и те не знаят как да го направят. Вие нямате причина да пишете такъв код!

Ако все пак, понякога вашата програмна логика изисква да игнорирате изключение, направете го, но сложете изрично коментар, с който обяснявате действията си. Ето едно типично изключение от това правило:

```
FileInputStream fis = null;
try {
   fis = new FileInputStream("file.bin");
   // Read the file here
```

```
} finally {
   try {
      fis.close();
   } catch(IOException e) {
      // Ignore the exception. The file is already closed
   }
}
```

В примера по-горе си позволяваме да игнорираме изключението, което може да възникне при затваряне на файл, защото ако файлът не може да бъде затворен, това означава, че той или е бил вече затворен от някой други, или не може да бъде затворен и няма смисъл да го мъчим повече. И в двата случая няма по-адекватна обработка на грешката, освен да я игнорираме.

Отпечатвайте съобщенията за грешка на конзолата само в краен случай!

В много книги за Java ще видите изключенията да се обработват като просто се отпечатат на конзолата:

```
try {
   // Some code here
} catch (Exception ex) {
   ex.printStackTrace();
}
```

Този код е изключително грешен, защото не дава възможност на извикващия метод да обработи грешката и да се възстанови от нея.

Представете си например нашия метод, който чете настройките на приложението от текстов файл. Ако възникне грешка, той би могъл да я отпечата на конзолата, но какво ще стане с извикващия метод? Той ще си помисли, че настройките са били успешно прочетени, нали?

Има едно много важно правило в програмирането:

Един метод или трябва да върши работата, за която е предназначен, или трябва да хвърля изключение.

Това правило можем да обясним в по-големи детайли: Един метод се пише, за да свърши някаква работа. Какво върши методът трябва да става ясно от неговото име. Ако не можем да дадем добро име на метода, значи той е прави много неща и трябва да се раздели на части. Ако един метод не може да свърши работата, за която е предназначен, той трябва да хвърли изключение. Например, ако имаме метод за сортиране на масив с числа, ако масивът е празен, методът или трябва да върне празен масив, или да

съобщи за грешка. Грешните входни данни трябва да предизвикват изключение, не грешен резултат! Например, ако се опитаме да вземем от даден стринг с дължина 10 символа подстринг от позиция 7 до позиция 12, трябва да получим изключение, не да вземем по-малко символи. Точно така работим методът **substring()** в класа **String**.

Не прихващайте всички изключения!

Една много често срещана грешка при работата с изключения е да се прихващат всички грешки, без оглед на техния тип. Ето един пример, при който грешките се обработват некоректно:

```
try {
   String fileContents = readFileContents("file.txt");
} catch (Throwable t) {
   System.out.println("File file.txt not found.");
}
```

В този код предполагаме, че имаме метод **readFileContents()**, който прочита текстов файл и го връща като **String**. Забелязваме, че **catch** блокът прихваща наведнъж всички изключения (независимо от типа им), не само **FileNotFoundException**, и при всички случаи отпечатва, че файлът не е намерен. Хубаво, обаче има ситуации, които са непредвидени. Например какво става, когато файлът е заключен от друг процес в операционната система. В такъв случай JVM ще се генерира **FileAccessDeniedException**, но съобщението за грешка, което програмата ще изведе, ще е грешно и подвеждащо. По същия начин, ако при отварянето на файла свърши паметта, ще се генерира съобщение **OurOfMemoryError**, но отпечатаната грешка ще е отново некоректна.

Изводът е от този пример е, че трябва да обработваме само грешките, които очакваме и за които сме подготвени. Останалите не трябва въобще да ги прихващаме.

Прихващайте само изключения, от които разбирате и знаете как да обработите!

Вече обяснихме, че даден метод трябва да прихваща само изключение, от които разбира, а не всички. Това е много важно правило, което непременно трябва да спазвате. Ако не знаете как да обработите даден exception, или не го прихващайте, или го обгърнете с ваш exception и го хвърлете по стека да си намери обработчик. Това е правилото: или добавете изключението в **throws** декларацията на вашия метод или го хванете и го опаковайте във ваше изключение и го хвърлете на извикващия метод. Иначе можете да стигнете до некоректна обработка на грешки, която може да доведе до много странни проблеми.

Ето пример за обработка на изключения чрез опаковане:

```
private void readSettings() {
   File file = new File("settings.txt");
   try {
      String fileContents = readFileContents(file);
      // Parse the contents and load settings here ...
   } catch (Exception ex) {
      throw new RuntimeException("Can not read the " +
         "settings from file " + file + ".");
   }
}
```

Ето пример за обработка на изключения чрез изхвърляне на изключенията, които не можем да обработим адекватно:

```
private void readSettings() throws IOException {
   File file = new File("settings.txt");
   String fileContents = readFileContents(file);
   // Parse the contents and load settings here ...
}
```

Упражнения

1. Да се намерят всички стандартни изключения от йерархията на **java.io.IOException**.
2. Да се намерят всички стандартни изключения от йерархията на **java.lang.RuntimeException**.
3. Да се намерят методите хвърлящи изключението **java.lang.IllegalArgumentException** и да се проучи причината за неговото хвърляне.
4. Да се намерят всички стандартни изключения от йерархията на **java.lang.IllegalArgumentException**.
5. Обяснете какво представляват изключенията, кога се използват и как се прихващат.
6. Обяснете ситуациите, при които се използва **try-finally** конструкцията.
7. Обяснете предимствата на използването на изключения.
8. Напишете метод, който приема като параметър име на текстов файл и прочита съдържанието му и го връща като **String**.
9. Напишете метод, който приема като параметър име на бинарен файл и прочита съдържанието на файла и го връща като масив от байтове. Напишете метод, който записва прочетеното съдържание в друг файл.
10. Потърсете информация в Интернет и дефинирайте собствен клас за изключение **FileParseException**. Вашето изключение трябва да съдържа

в себе си името на файл, който се обработва и номер на ред, в който възниква изключението. Добавете подходящи конструктори за вашето изключение. Напишете програма, която чете от текстов файл числа. Ако при четенето се стигне до ред, който не съдържа число, хвърлете вашия exception и го обработете в извикващия метод.

Решения и упътвания

1. Използвайте среда за разработка Eclipse и вградената възможност за разглеждане на йерархии от класове. Отворете класа **java.io.IOException** като използвате вграденото търсене на класове – [Ctrl-Shift-T] и натиснете [F4], за да отворите визуализатора на йерархии.
2. Разгледайте упътването за предходната задача.
3. Използвайте среда за разработка Eclipse и възможността за търсене на използванията на даден клас. Отворете класа **java.lang.IllegalArgumentException**, позиционирайте курсора на дефиницията на класа и потърсете къде се използва като натиснете [Ctrl-H].
4. Разгледайте упътването за задача 1.
5. Използвайте информацията от началото на главата.
6. При затруднения използвайте информацията от главата.
7. При затруднения използвайте информацията от главата.
8. Прочетете файла ред по ред с класа **java.util.Scanner** и добавяйте редовете в **StringBuilder**. Декларирайте в сигнатурата на метода, че изхвърля **IOException** и не обработвайте никакви изключения в него.
9. Малко е вероятно да напишете коректно този метод от първи път без чужда помощ. Първо прочетете в Интернет как се работи с бинарни потоци. След това следвайте препоръките по-долу за четенето на файла:
   - Декларирайте в сигнатурата на метода, че изхвърля **IOException** и не обработвайте никакви изключения в него.
   - Използвайте за четене **FileInputStream**, а прочетените данни записвайте в **ByteArrayOutputStream**.
   - Внимавайте с метода за четене на байтове **read(byte[] buffer, int offset, int count)**. Този метод може да прочете по-малко байтове, отколкото сте заявени. Колкото байта прочетете от входния поток, толкова трябва да запишете. Трябва да организирате цикъл, който завършва при връщане на стойност -1.
   - Използвайте **try-finally**, за да затваряте потоците.

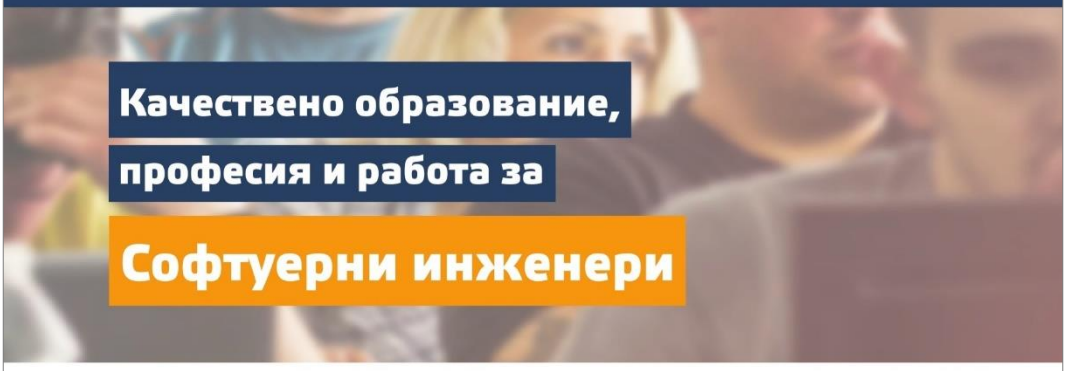
   Записването на масив от байтове във файл е далеч по-проста задача. Отворете **FileOutputStream** и запишете в него масива. Изхвърлете чрез

throws всички изключения от метода. Използвайте **try-finally**, за да затваряте потоците.

Накрая тествайте с някой ZIP архив. Ако програмата ви работи некоректно, ще счупите структурата на архива и ще се получава грешка при отваряне.

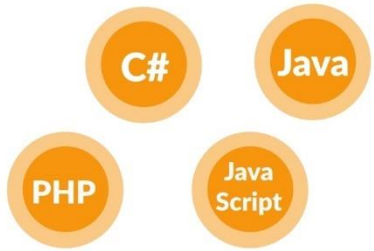
10. Наследете класа **Exception** и добавете подходящ конструктор, примерно **FileParseException(string msg, String filename, int line)**. След това ползвайте вашето изключение както ползвате всички други изключения, които познавате. За четене на файла ползвайте **FileReader** и от него създайте **BufferedReader**.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 13. Символни низове

Автор

Марио Пешев

В тази тема...

В настоящата тема ще се запознаем със символните низове: как са реализирани те в Java и по какъв начин можем да обработваме текстово съдържание. Ще прегледаме различни методи за манипулация на текст; ще научим как да извличаме поднизове по зададени параметри, как да търсим за ключови думи, както и да отделяме един низ по разделители. Ще се запознаем с методи и класове за по-елегантно и стриктно форматиране на текстовото съдържание на конзолата, с различни методики за извеждане на числа, а също и с извеждането на отделни компоненти на текущата дата. Накрая ще предоставим полезна информация за регулярните изрази и ще научим по какъв начин да извличаме данни, отговарящи на определен шаблон.

Символни низове

В практиката често се налага обработката на текст: четене на текстови файлове, търсене на ключови думи и заместването им в даден параграф, валидиране на входни потребителски данни и др. В такива случаи можем да запишем текстовото съдържание, с което ще боравим, в символни низове, и да го обработим с помощта на езика Java.

Какво е символен низ (стринг)?

Символният низ е последователност от символи, записана на даден адрес в паметта. Помните ли типа **char**? В променливите от тип **char** можем да запишем само 1 символ. Когато е необходимо да обработваме повече от един символ, на помощ идват стринговете.

В Java всеки символ има пореден номер в **Unicode** таблицата. Unicode е стандарт, създаден в края на 80-те и началото на 90-те години с цел съхраняването на различни типове текстови данни. Предшественикът му ASCII позволява записването на едва 128 или 256 символа (ASCII стандарт със 7-битова или 8-битова таблица). За съжаление, това често не удовлетворява нуждите на потребителя – тъй като в 128 символа могат да се поберат само цифри, малки и главни латински букви и някои специални

знаци. Когато опре до работа с текст на кирилица или друг специфичен език (например азиатски или африкански), 128 символа са напълно недостатъчни. Ето защо Java използва 16-битова кодова таблица за символи. С помощта на знанията ни за бройните системи и представянето на информацията в компютрите, можем да сметнем, че кодовата таблица съхранява 2^16 = 65536 символа. Някои от символите се кодират по специфичен начин, така че е възможно използването на 2 символа от Unicode таблицата за създаване на нов символ – така получените знаци надхвърлят 100 000.

Класът java.lang.String

Класът **java.lang.String** позволява обработка на символни низове в Java. Работата със **String** ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции. Пример за декларация на символен низ:

```
String greeting = "Hello, Java";
```

Декларирахме променливата **greeting** от тип **String**, която има съдържание "**Hello, Java**". Представянето на съдържанието в символния низ изглежда по подобен начин:

H	e	l	l	o	,		J	a	v	a

Вътрешното представяне на класа е съвсем просто – масив от символи. По принцип ние можем да избегнем използването на класа, като декларираме променлива от тип **char[]** и запълним елементите на масива символ по символ. Недостатъците на това обаче са няколко:

1. Запълването на масива става символ по символ, а не наведнъж.
2. Трябва да знаем колко дълъг ще е текстът, за да сме наясно дали ще се побере в заделеното място за масива.
3. Обработката на текстовото съдържание става ръчно.

Класът String – универсално решение?

Използването на **String** не е идеално и универсално решение – понякога е уместно използването на други символни структури.

В Java съществуват и други класове за обработка на текст – с някои от тях ще се запознаем по-нататък в главата.

Класът **String** има важна особеност – последователностите от символи, записани в променлива от класа, са **неизменими (immutable)**. Веднъж записано, съдържанието на променливата не се променя директно - ако опитаме да променим стойността, тя ще бъде записана на ново място в динамичната памет, а променливата ще започне да сочи към него.

Типът **String** е по-особен от останалите типове данни. Той е клас и спазва принципите на обектно-ориентираното програмиране: стойностите се записват в динамичната памет, а променливите пазят препратка към паметта (референция към обект в динамичната памет). От друга страна, **String** променливите са неизменими. Ако няколко променливи сочат към една и съща област в паметта с дадена стойност, тази стойност не може да бъде директно променена. Промяната ще се отрази само на променливата, чрез която е редактирана стойността, тъй като това ще създаде нова стойност в динамичната памет и ще насочи въпросната променлива към нея, докато останалите променливи ще сочат на старото място.

Символни низове – прост пример

Използването на променливи от тип **String** изглежда по следния начин:

```
String msg = "Stand up, stand up, Balkan superman.";

System.out.printf("msg = \"%s\"%n", msg);
System.out.printf("msg.length() = %d%n", msg.length());

for (int i = 0; i < msg.length(); i++) {
   System.out.printf("msg[%d] = %c%n", i, msg.charAt(i));
}
```

В посочения фрагмент от код виждаме декларация на променливата **s** и задаването на стойност:

```
Stand up, stand up, Balkan superman.
```

Обърнете внимание на стойността на стринга – кавичките не са част от текста, а ограждат стойността му.

Ето как изглежда резултатът от изпълнението на горния пример (със съкращения):

```
msg = "Stand up, stand up, Balkan superman."
msg.length() = 36
msg[0] = S
msg[1] = t
msg[2] = a
msg[3] = n
msg[4] = d
...
```

Escaping при символните низове

Ако искаме да използваме кавички в съдържанието, тогава трябва да поставим наклонена черта преди тях за указание на компилатора.

```
String quote = "Book’s title is \"Intro to Java\"";
```

Съдържанието на променливата **quote** е:

```
Book's title is "Intro to Java"
```

Кавичките този път са част от текста. В променливата те са добавени чрез поставянето им след екраниращия знак (**escaping character**) обратна наклонена черта (****). По този начин компилаторът разбира, че кавичките не служат за начало или край на символен низ, а са част от данните. Наклонената черта се използва за символи, които играят специална роля в текста (в случая кавичките) или за дефиниране на действие, което не може да се изрази със символ. Пример за втория случай са обозначаването на символ за нов ред (**\n**), табулация (**\t**), избор на символ по неговия Unicode (**\uXXXX**, където с **X** се обозначава кодът) и др.

Деклариране на символен низ

Можем да декларираме променливи от тип символен низ чрез класа **java.lang.String**:

```
String str;
```

Декларацията на символен низ представлява декларация на променлива от класа **String**. Това не е еквивалентно на създаването на променлива и заделянето на памет за нея! С декларацията уведомяваме компилатора, че ще използваме променлива **str** и очакваният тип за нея е **String**. Ние не създаваме променливата в паметта и тя все още не е достъпна за обработки (има стойност **null**, което означава липса на стойност).

Създаване и инициализиране на символен низ

За да може да обработваме декларираната променлива, трябва да я създадем и инициализираме. Създаването на променлива на клас (познато още като **инстанциране**) е процес, свързан със заделянето на област в динамичната памет. Преди да зададем конкретна стойност на символния низ, стойността му е **null**. Това може да бъде объркващо за начинаещия програмист: неинициализираните променливи от типа **String** не съдържат празни стойности, а специалната стойност **null** – и опитът за манипулация на такъв стринг ще генерира грешка (изключение за достъп до липсваща стойност **NullPointerException**)!

Можем да инициализираме променливи по 3 начина:

1. Чрез задаване на символна константа
2. Чрез присвояване стойността на друг символен низ

3. Чрез предаване стойността на операция, връщаща символен низ

Задаване на символна константа

Задаването на символна константа (**литерал**) означава предаване на предефинирано текстово съдържание на променлива от тип **String**. Използваме такъв тип инициализация, когато знаем стойността, която трябва да се съхрани в променливата. Пример за задаване на константа е:

```
String website = "http://academy.devbg.org";
```

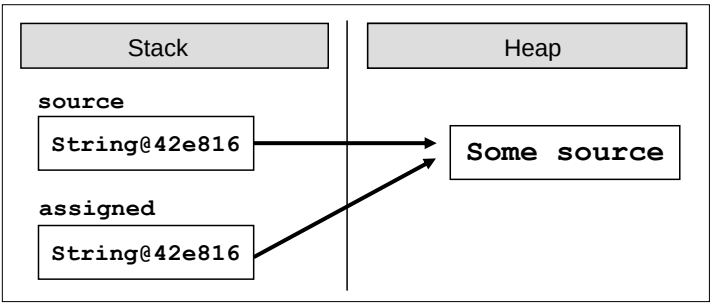
Тук създаваме променливата **website** и ѝ задаваме константна стойност, която е приемлива за типа **String** (символен литерал).

Присвояване стойността на друг символен низ

Присвояването на стойността е еквивалентно на насочване на **String** променлива към друга променлива от същия тип. Пример за това е следният фрагмент:

```
String source = "Some source";
String assigned = source;
```

В примера откриваме съществуващата променлива **source**, която има присвоена стойност. Присвояването на въпросната стойност на друга променлива е във втория ред - променливата **assigned** приема стойността на **source**. Тъй като класът **String** е референтен тип, на по-ниско ниво "**Some source**" е записано в динамичната памет (heap, хийп), сочено от първата променлива. На втория ред пренасочваме променливата **assigned** към същото място, към което сочи другата променлива. Така двата обекта имат една и съща стойност:



Промяната на коя да е от променливите обаче ще се отрази само и единствено на нея, поради неизменността на типа **String**. Това не се отнася за останалите референтни типове, които не са неизменни (immutable), защото при тях промените се нанасят на адреса в паметта и всички референции сочат към променения обект.

Предаване стойността на операция, връщаща символен низ

Третият вариант за инициализиране на символен низ е предаването на стойността на израз или операция, която връща същия резултат. Това може да бъде резултат от метод, който валидира данни; събиране на стойностите на няколко константи и променливи, преобразуване на съществуваща променлива и др. Пример за израз, връщащ символен низ:

```
String email = "some@email.bg";
String info = "My mail is: " + email + ".";
// My mail is: some@email.bg.
```

Променливата **info** е създадена от съединяването (**concatenation**) на литерали и променлива.

Четене и печатане на конзолата

Нека сега разгледаме как можем да четем символни низове, въведени от потребителя, и как можем да печатаме символни низове на стандартния изход (на конзолата).

Четене на символни низове

Четенето на символни низове може да бъде осъществено чрез методите на познатия ни клас **java.util.Scanner**:

```
Scanner input = new Scanner(System.in);
String name = input.nextLine();
```

На първо време създаваме инстанция от класа **Scanner**, подавайки като параметър входния поток **System.in**. След това, използвайки създадения обект, прочитаме от конзолата входните данни чрез метода **nextLine()**. След натискане на клавиша [Enter] от потребителя, променливата **name** ще съдържа въведеното име от клавиатурата.

Какво можем да правим, след като променливата е създадена и в нея има стойност? Например да я използваме в изрази с други символни низове, да я подаваме като параметър на методи, да я записваме в текстови документи и др. На първо време, можем да я изведем на конзолата, за да се уверим, че данните са прочетени коректно.

Отпечатване на символни низове

Извеждането на данни се извършва чрез изходния поток **System.out**:

```
System.out.println("Your name is: " + name);
```

Използвайки метода **println(…)** извеждаме съобщението: **Your name is:**, придружено със стойността на променливата **name**. След края на съобщението се добавя символ за нов ред, като следващото съобщение ще бъде изведено на следващия ред на конзолата. Ако искаме да избегнем символа за нов ред и съобщенията да се извеждат на един и същ, тогава прибягваме към метода **print(…)**.

В случай, че ни трябва по-прецизен форматиран изход, на помощ идва методът **printf(…)**:

```
System.out.printf("Hello, %s, have a nice reading!", name);
```

Ако стойността на променливата **name** е "**Mario Peshev**", то резултатът от изпълнението на горния ред ще е **Hello, Mario Peshev, have a nice reading!**

Операции върху символни низове

След като се запознахме със семантиката на символните низове, как можем да ги създаваме и извеждаме, следва да се научим как да боравим с тях и да ги обработваме. Езикът Java ни дава набор от готови функции, които ще използваме за манипулация над стринговете.

Сравняване на низове по азбучен ред

Има множество начини за сравнение на символни низове. В зависимост от това какво точно ни е необходимо в конкретния случай, може да се възползваме от различни възможности на класа **String**.

Сравнение за еднаквост

Ако условието изисква да сравним два символни низа и да установим дали стойностите им са еднакви или не, удобни методи са **equals(…)** и **equalsIgnoreCase(…)**. Двата метода връщат булев резултат със стойност **true**, ако низовете имат еднакви стойности, и **false**, ако те са различни. Първата функция проверява за равенство на стойностите на променливите, като прави разлика между малки и главни букви. Т.е. сравняването на "Java" и "JAVA" с метода **equals(…)** ще върне стойност **false**. В практиката често ще ни интересува самото съдържание, без значение от регистъра (**casing**) на буквите. Използването на метода **equalsIgnoreCase(…)** в горния пример би игнорирал разликата между малки и главни букви и ще върне стойност **true**.

```
String word1 = "Java";
String word2 = "JAVA";
System.out.println(word1.equals(word2)); // false
System.out.println(word1.equalsIgnoreCase(word2)); // true
```

Сравнение на низове по азбучен ред

Дотук добре, но как ще установим лексикографската подредба на няколко низа? Ако искаме да сравним две думи и да получим данни коя от тях е преди другата, според азбучния ред на буквите в нея, на помощ идват **compareTo(…)** и **compareToIgnoreCase(…)**. Двата метода ни дават възможност да сравним стойностите на два символни низа, като установим лексикографския им ред.

Връщайки се на темата за кодировката на символите, си припомняме, че всеки символ има свой уникален номер в Unicode таблицата. Например главната латинска буква "B" има стойност 66, докато главната "E" – 69. Методът **compareTo(…)** сравнява 2 символни низа за равенство или различие. За да бъдат два низа с еднакви стойности, то те трябва да имат една и съща дължина (брой символи) и символите да бъдат еднакви и подредени в един и същ ред. Низовете "give" и "given" са различни, защото имат различна дължина. "near" и "fear" се различават по първия си символ, а "stop" и "post" имат едни и същи символи и дължина, но в различен ред - което отново ги прави различни.

Обобщавайки поведението на метода, можем да приемем, че **compareTo(…)** връща положително число, отрицателно число или 0 в зависимост от лексикографската подредба.

За да не бъдем голословни, нека разгледаме няколко примера:

```
String score = "sCore";
String scary = "scary";
System.out.println(score.compareToIgnoreCase(scary)); // 14
System.out.println(scary.compareToIgnoreCase(score)); // -14
System.out.println(scary.compareTo(score)); // 32
```

За примера ще използваме променливи със стойности "sCore" и "scary". Първият експеримент е извикването на метода **compareToIgnoreCase(…)** на низа **score**, като подаден параметър е променливата **scary**. Тъй като методът игнорира регистъра за малки и главни букви, първите 2 символа и от двата низа връщат знак за равенство. Различието се открива едва в третия символ, който в първия низ е "o", а във втория: "a". Тогава изваждаме кода на параметъра от кода на променливата, за която е извикан методът. Крайният резултат е 14 (кодът на 'o' е 111, кодът на 'a' е 97; 111-97 = 14). Извикването на същия метод с разменени места на променливите връща -14, защото тогава отправната точка е низът **scary** и кодовете се изваждат в обратен ред.

Последният тест е с метода **compareTo(…)** – тъй като той прави разлика между главни и малки букви, разлика откриваме още във втория символ на двата низа. В променливата **scary** символът "c" има код 99, в **score** главното "C" е 67 и връщаният резултат е 32.

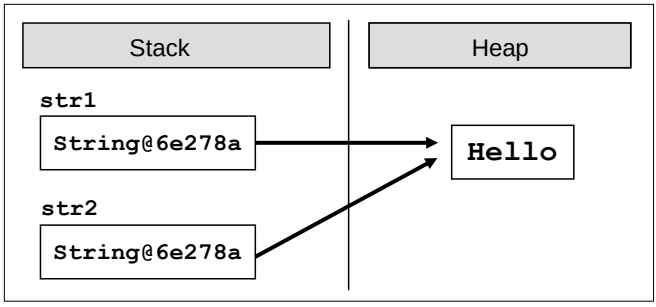
Защо операторите == и != не работят за низове?

По-любознателните от вас може вече да са се запитали защо операторите за равенство и различие не работят при работа със символни низове? Причината е тяхната логика в света на обектно-ориентираното програмиране. Когато работим със стойностни типове (цели числа, символи, числа с плаваща запетая), тогава операторите сравняват стойностите на променливите. Тъй като символните низове в Java са реализирани с класове, тук влизат правилата за сравняване на препратки към паметта, известни още като референции или указатели. Тогава сравняването на две променливи **str1** и **str2** няма да сравнява техните стойности, а дали те сочат към една и съща област в динамичната памет.

Ще прегледаме няколко примера за използването на тези два оператора с променливи от тип символни низове:

```
String str1 = new String("Hello");
String str2 = str1;
System.out.println((str1==str2)); // true
```

Сравнението на низовете **str1** и **str2** връща стойност **true**. Това е очакван резултат, тъй като насочваме променливата **str2** към мястото в динамичната памет, което е запазено за променливата **str1**. Така двете променливи имат един и същ адрес и проверката за равенство минава успешно. Ето как изглежда паметта с двете променливи:

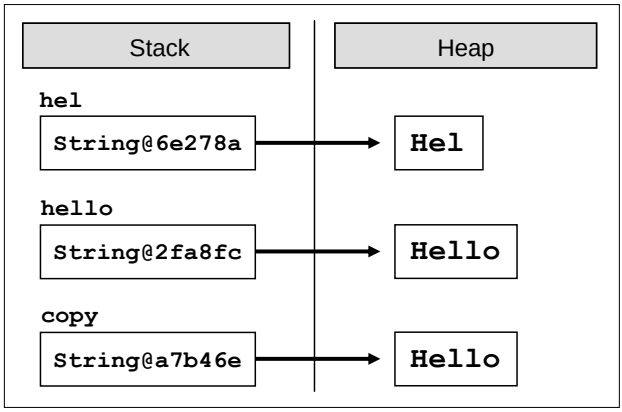


Да прегледаме сега един друг пример:

```
String hel = "Hel";
String hello = "Hello";
String copy = hel + "lo";
System.out.println(copy == hello); // false
```

Сравнението е между низовете **hello** и **copy**. Първата променлива директно приема стойността "Hello". Втората получава стойността си след съединяването на променлива и литерал, като крайният резултат е еквивалентен на стойността на първата променлива. Сравнението обаче връща стойност **false** – защото двете променливи сочат към различни

области от паметта (макар и с еднакво съдържание). Ето как изглежда паметта в този момент:



Когато две променливи от тип обект (в частност стрингове) сочат към различни места в паметта, те се считат за различни обекти.

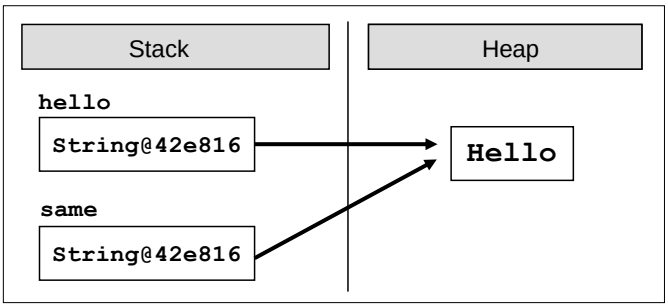
Сравняването на низове с оператора == в Java е груба грешка, защото този оператор сравнява адресите на низовете, не техните стойности! За сравняване на низове използвайте методите `equals()` / `equalsIgnorecase()` и `compareTo()` / `compareToIgnoreCase()` и проверявайте за изрично за `null`, защото извикването на `equals()` върху `null` стойност предизвиква `NullPointerException`.

Любопитен обаче е следният случай:

```
String hello = "Hello";
String same = "Hello";
System.out.println(same == hello); // true
```

Оптимизация на паметта при символни низове

Създаваме първата променлива със стойност "**`Hello`**". Създаваме и втората променлива със същата стойност. Логично е при създаването на променливата **`hello`** да се задели място в динамичната памет, да се запише стойността и променливата да сочи към въпросното място. При създаването на **`same`** също би трябвало да се създаде нова област, да се запише стойността и да се насочи препратката. Истината обаче е, че съществува оптимизация във виртуалната машина, която спестява създаването на дублиращи символни низове в паметта. Тази оптимизация се нарича **strings interning** (интерниране на низовете) и благодарение на нея двете променливи в паметта се записват по следния начин:

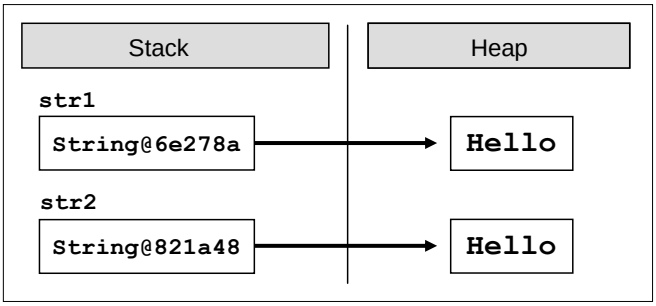


Когато инициализираме променлива от тип **String**, динамичната памет се обхожда и се прави проверка дали такава стойност вече съществува. Ако съществува, новата променлива просто започва да сочи към нея. Ако не, заделя се място, стойността се записва и референцията препраща към новата област. Това е възможно, защото стринговете в Java са неизменни и няма опасност за промяна на областта, сочена от няколко променливи едновременно.

Все пак, ако не искаме да използваме оптимизацията, можем изрично да създадем нови обекти от тип **String** и да зададем техните стойности:

```
String str1 = new String("Hello");
String str2 = new String("Hello");
System.out.println((str1==str2)); // false
```

Извикването на конструктора на класа директно заделя място в паметта за променливата и вмъква в него съдържанието й. Връщаният резултат е **false**, тъй като всяка променлива сочи на отделно място в паметта:



Операции за манипулация на символни низове

След като се запознахме с основите на символните низове и тяхната структура, идва ред на манипулацията им. Ще прегледаме слепването на текстови низове, търсене в съдържанието им, извличане на поднизове и други операции, които ще ни послужат при решаване на реални проблеми.

Символните низове са неизменими! Всяка промяна на променлива от тип `String` създава нов низ, в който се записва резултатът. Така че операциите, които прилагате на символните низове, връщат като резултат препратка към получения резултат.

Примерите за обработка на символни низове по-долу използват променливи от тип **`String`**. Както вече споменахме, промените на обектите от класа **`String`** връщат референция към новата област от паметта, в която се намира резултатът.

Възможна е и обработката на символни низове без създаването на нови обекти в паметта при всяка корекция. За целта е създаден класът **`StringBuilder`**, с който ще се запознаем по-долу.

Долепване на низове (конкатенация)

Долепването на символни низове и получаването на нов, резултатен низ, се нарича конкатенация. То може да бъде извършено по 2 начина: чрез метода **`concat(…)`** или с оператора **`+`**, или **`+=`**.

Пример за използване на функцията **`concat(…)`**:

```
String greet = "Hello, ";
String name = "reader!";
String result = greet.concat(name);
```

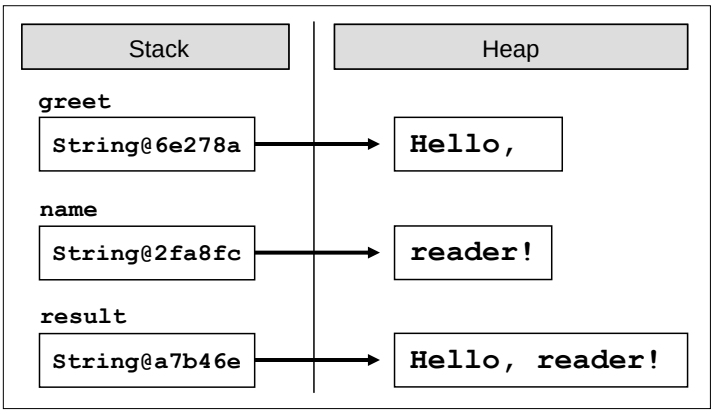
Извиквайки метода, ще долепим променливата **`name`**, която е подадена като аргумент, към променливата **`greet`**. Резултатният низ ще има стойност "**`Hello, reader!`**".

Вторият вариант за конкатенация е чрез операторите **`+`** и **`+=`**. Горния пример може да реализираме без проблем и по двата начина, например:

```
String greet = "Hello, ";
String name = "reader!";
String result = greet + name;
```

В паметта тези променливи ще изглеждат както на фигурата по-долу.

Обърнете внимание, че всички долепвания до низове не променят съществуващите променливи, а връщат нова променлива като резултат. Ако опитаме да долепим 2 стринга, без да ги запазим в променлива, промените нямат да бъдат съхранени.



В случая може да искаме да добавим някаква стойност към променливата **result**. С познатите ни оператори възможно решение е следното:

```
result = result + " How are you?";
```

За да си спестим повторното писане на декларираната по-горе променлива, можем да използваме оператора **+=**:

```
result += " How are you?";
```

И в двата случая резултатът ще бъде един и същ: "**Hello, reader! How are you?**".

Към символните низове можем да долепим и други данни, които могат да бъдат представени в текстов вид. Възможна е конкатенацията с числа, символи, дати и др. Ето един пример:

```
String msg = "The number of the beast is: ";
int beastNum = 666;
String result = msg + beastNum;
// The number of the beast is: 666
```

Както виждаме от горния пример, няма проблем да съединяваме символни низове с други данни, които не са от тип **String**. Нека прегледаме още един, пълен пример за слепването на символни низове:

```
public class UserInfoExtractor {
   public static void main(String[] args) {
      String firstName = "Svetlin";
      String lastName = "Nakov";
      String fullName = firstName + " " + lastName;
      System.out.println(fullName);
      // Svetlin Nakov
```

```
        int age = 28;
        String nameAndAge = "Name: " + fullName + "\nAge: " + age;
        System.out.println(nameAndAge);
        // Name: Svetlin Nakov
        // Age: 28
    }
}
```

Търсене на низ в друг низ

Когато имаме символен низ със зададено съдържание, често се налага да обработим само част от стойността му. За да автоматизираме процеса, можем да претърсваме даден стринг за определени ключови думи.

Java платформата ни предоставя 2 метода за търсене на низове: **`indexOf(…)`** и **`lastIndexOf(…)`**. Те претърсват даден символен низ и проверяват дали подаденият като параметър подниз се среща в съдържанието му. Връщаният резултат на методите е цяло число. Ако резултатът е неотрицателна стойност, тогава това е позицията, на която е открит първият символ от подниза. Ако методът върне стойност **`-1`**, това означава, че поднизът не е открит. Напомняме, че в Java индексите на символите в низовете започват от 0.

Търсене в символен низ – пример

Ето и един пример за използване на метода **`indexOf(…)`**:

```
String book = "Introduction to Java book";
int index = book.indexOf("Java");
System.out.println(index); // index = 16
```

В примера променливата **`book`** има стойност "Introduction to Java book". Търсенето на подниза "Java" в горната променлива ще върне стойност 16, защото поднизът е открит в стойността на отправната променлива и първият символ "**`J`**" от търсената дума се намира на 16-та позиция.

Методите **`indexOf(…)`** и **`lastIndexOf(…)`** претърсват съдържанието на текстова последователност, но в различна посока. Търсенето при първата функция започва от началото на низа към неговия край, а при втората функция – отзад-напред. Когато се интересуваме от първия срещнат резултат, тогава използваме **`indexOf(…)`**. Ако искаме да претърсваме низа от неговия край (например за откриване на последната точка в името на даден файл или последната наклонена черта в URL адрес), уместно решение е **`lastIndexOf(…)`**.

Понякога искаме да открием всички срещания на даден подниз в текущия низ. Използването на двата метода само с 1 подаден аргумент за търсен низ не би ни свършило работа, защото винаги ще връща само първото срещане

на подниза. Ето защо е възможно подаването на втори параметър за индекс, който посочва началната позиция, от която започва търсенето.

Всички срещания на дадена дума – пример

Ето един пример за използването на **`indexOf (…) по`** дадена дума и начален индекс: откриване на всички срещания на думата "Java" в даден текст:

```
String quote = "The main subject in the \"Intro Java\"" +
   " book is Java for Java newbies.";
int index = quote.indexOf("Java");
while(index != -1) {
   System.out.println("Java found on index: " + index);
   index = quote.indexOf("Java", index + 1);
}
```

Първата стъпка е да направим търсене за ключовата дума "Java". Ако думата е открита в текста (т.е. връщаната стойност е различна от -1), извеждаме я на конзолата и продължаваме търсенето надясно от позицията, на която сме открили думата, увеличена с 1. Повтаряме действието, докато **`indexOf(…)`** върне стойност -1.

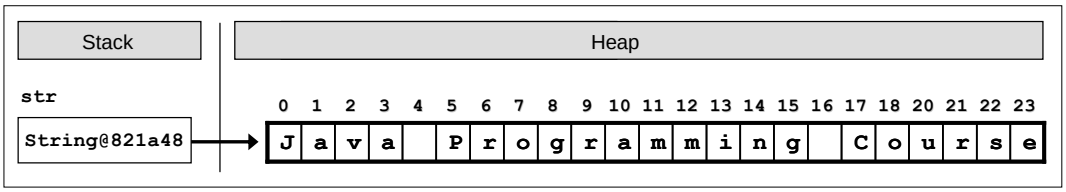
Забележка: Ако на последния ред пропуснем задаването на начален индекс, то търсенето винаги ще започва отначало и ще връща една и съща стойност. Това ще доведе до безкраен цикъл на приложението ни. Ако пък търсим директно от индекса, без да увеличаваме с единица, ще попадаме отново на последния резултат, чийто индекс сме записали. Ето защо правилното търсене на следващ резултат е с аргумент **`index + 1`**. За **`lastIndexOf(…)`**, аналогично, тъй като търсенето е в обратен ред, индексът се намалява с единица.

Подробно търсене с indexOf(…) – пример

Нека прегледаме още един подробен пример за търсенето на отделни символи и символни низове в текст:

```
String str = "Java Programming Course";

int index = str.indexOf("Java"); // index = 0
index = str.indexOf("Course"); // index = 17
index = str.indexOf("COURSE"); // index = -1
// indexOf() is case sensetive. -1 means "not found"
index = str.indexOf("ram"); // index = 9
index = str.indexOf("r"); // index = 6
index = str.indexOf("r", 7); // index = 9
index = str.indexOf("r", 10); // index = 20
```

Ето как изглежда в паметта символният низ, в който търсим:



Ако обърнем внимание на третата проверка, ще забележим, че търсенето на думата "**COURSE**" в текста връща резултат -1, т.е. няма съответствие. Въпреки че думата се намира в текста, тя е написана с различен регистър на буквите. Методите **indexOf(…)** и **lastIndexOf(…)** правят разлика между малки и главни букви. Ако искаме да игнорираме тази разлика, можем да запишем текста в нова променлива и да го превърнем към такъв с изцяло малки или изцяло главни букви, след което да извършим търсене в него, независещо от регистъра на буквите.

Извличане на част от низ

За момента можем само да проверим дали даден подниз се намира в нашия текст и на коя позиция го откриваме. Как обаче да извлечем част от низа в отделна променлива?

Решението на проблема ни е методът **substring(…)**. Използвайки въпросната функция, можем да извлечем даден подниз по зададени начална и крайна позиция в текста. Ако крайната позиция бъде пропусната, то подтекстът ще бъде копиран от началната позиция до края на символния низ.

Използването на дефиницията на метода с начален и краен индекс изглежда по този начин:

```
String path = "C:\\Pics\\Rila2008.jpg";
String filename = path.substring(8, 16);
// filename = "Rila2008"
```

Променливата, която манипулираме, е **path**. Тя съдържа пътя до файл от файловата ни система. За да присвоим името на файла на нова променлива, използваме **substring(8, 16)** и взимаме последователността от символи, намираща се на позиции от 8 до 16. Символът на последната позиция (в случая 16) не се записва в подниза!

Извикването на метода `substring(индекс1, индекс2)` **извлича подниз на дадена променлива, който се намира между** `индекс1` **и** `(индекс2 - 1)` **включително. Символът на посочената позиция –** `индекс2` **– не се взима предвид! Например, ако посочим** `substring(5, 10)`**, ще бъдат извлечени символите между индекс 5 и 9 включително, а не между 5 и 10!**

Можем да разглеждаме променливата **path** като масив от символи, за да придобием по-ясна представа за местоположението и броя символи:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
C	:	\\	P	i	c	s	\\	R	i	l	a	2	0	0	8	.	j	p	g

Придържайки се към схемата, извикваният метод трябва да запише символите от 8 до 15 включително (тъй като последният индекс не се включва), а именно "**Rila2008**".

Как бихме могли да изведем името на файла и неговото разширение? Тъй като знаем как се записва път във файловата система, можем да процедираме по следния план:

- търсим последната обратна наклонена черта в текста;
- записваме позицията на последната наклонена черта;
- извличаме подниза от **получената позиция + 1**.

Да вземем отново за пример познатия ни **path**. Ако нямаме информация за съдържанието на променливата, но знаем, че е файлов път, може да се придържаме към горната схема:

```
String path = "C:\\Pics\\Rila2008.jpg";
int index = path.lastIndexOf("\\"); // index = 7
String fullName = path.substring(index + 1);
// fullname = "Rila2008.jpg"
```

Разцепване на низ по разделител

Един от най-гъвкавите методи за работа със символни низове е **split(…)**. Той ни дава възможност да разцепваме един стринг по разделител или група разделители. Например, можем да обработваме променлива, която има следното съдържание:

```
String listOfBeers = "Amstel, Zagorka, Tuborg, Becks.";
```

Как можем да отделим всяка една бира в отделна променлива или да запишем всички бири в масив? На пръв поглед може да изглежда трудно – трябва да търсим с **indexOf(…)** за специален символ, след това да отделяме подниз със **substring(…)**, да итерираме всичко това в цикъл и да записваме резултата в дадена променлива.

Разделяне на низа по множество от разделители – пример

Има и доста по-лесен и гъвкав начин да разрешим проблема:

```
String[] beersArr = listOfBeers.split("[ ,.]");
```

Използвайки вградената функционалност на **split(…)**, ще разделим съдържанието на даден низ по разделителите, които са подадени като аргумент на метода. В квадратни скоби са изредени всички разделители, които искаме да използваме за отправна точка при разделянето на думите. Всички поднизове, между които присъстват интервал, запетая или точка, ще бъдат отделени и записани в масива **beersArr**.

Все пак, ако обходим масива и изведем елементите му един по един, резултатите ще бъдат: "Amstel", "", "Zagorka", "", "Tuborg", "" и "Becks". Получаваме 7 резултата, вместо очакваните 4. Причината е, че при разделянето на текста се откриват 3 подниза, които съдържат 2 разделителни символа един до друг (например запетая, последвана от интервал). В този случай празният низ между двата разделителя също е част от връщания резултат.

Как да премахнем празните елементи?

Ако искаме да игнорираме празните низове, възможно разрешение е да правим проверка при извеждането им:

```
for(String beer : beersArr) {
   if(!beer.equals("")) {
      System.out.println(beer);
   }
}
```

Този подход обаче няма да премахне празните низове от масива. Затова можем да променим аргумента, който подаваме на метода **split(…)**, като добавим знака **+**:

```
String[] beersArr = listOfBeers.split("[ ,.]+");
```

След тази промяна масивът **beersArr** ще съдържа 4 елемента – четирите думи от променливата **listOfBeers**. Добавяйки плюс към заградените символи, ние инструктираме метода **split(…)** да работи по следния начин: "**Върни всички поднизове от променливата, които са разделени от интервал, запетая или точка. Ако срещнеш два или повече съседни разделителя, считай ги за един**".

Обяснение на метода split(...)

Методът **split(…)** действително е един от най-комплексните и гъвкави методи за работа със символни низове. Неговата функционалност се дължи на факта, че той приема като аргумент **регулярен израз (regular expression)**. Регулярният израз е символен низ, представящ множества или подмножества. Пример за множества и подмножества са всички малки букви; всички цифри; главните латински букви от 'A' до 'M'; всички малки и главни латински и кирилски символи, и др. Обединяването на множества позволява по-прецизна обработка на текстови данни: извличане на

определени ресурси от текстове, търсене на телефонни номера, откриване на електронна поща в текст, разделяне на всички думи в едно изречение и т.н.

Квадратните скоби служат за изреждане на отделни символи, подмножества и множества. Например, за да дефинираме множество от всички малки латински букви, регулярният израз ще изглежда така:

```
String small = "[a-z]";
```

По този начин посочваме множеството от символи, намиращи се в интервала от 'a' до 'z'. Ако искаме да обхванем всички малки и главни букви на латиница и кирилица, можем да дефинираме следното множество:

```
String allLetters= "[a-zA-Za-яA-Я]";
```

В сила са и операндите OR, AND и NOT за работа с множества. Можем, например, да разделим даден текст по всички символи, които не са цифри:

```
String nan = "[^0-9]";
```

Възможно е изграждането на сложни регулярни изрази, изградени от много обединения и сечения на множества. Те могат да бъдат използвани за задаване на набор от разделители при използването на **split(…)**, за търсене на подниз, съвпадащ с определен шаблон, или за заместване на един низ с друг по определени критерии.

Регулярни изрази – полезни конструкции

Съществуват предефинирани класове, които обобщават често използвани множества. Някои от тях са:

- **.** (символът **точка**) – обхваща всички възможни символи (може да прихваща или не обозначенията за нов ред)
- **\d** – обхваща всички цифри (еквивалентно на **[0-9]**)
- **\D** – обхваща всички символи, които не са цифри (еквивалентно на **[^0-9]**)
- **\s –** знак за интервали: **[\t\n\x0B\f\r]**
- **\S –** всички знаци, освен тези за интервали: **[^\s]**
- **\w –** всички символи, считани за дума: **[a-zA-Z_0-9]**
- **\W –** еквивалентно на **[^\w]**

Когато използвате предефинираните класове в Java, не забравяйте, че обратната наклонена черта е екраниращ знак! По тази причина е необходимо да добавяме още една обратна наклонена черта, за да използваме предефинираните класове.

Използване на предефинирани класове – пример

Ако имаме символен низ, представящ опростен каталог на продуктите в магазин за техника, то данните в него ще бъдат представени като двойки име на продукт: цена на продукта в лева. Ще създадем проста програма, използваща метода **split(…)**, която първо извежда на екрана само продуктите от каталога (без придружаващите ги цени), а след това изписва единствено списък от цените.

```
public class CatalogApplication {
   public static void main(String[] args) {
      String catalog =
         "MicrowaveOven: 170, \n" +
         "AudioSystem: 125, \n" +
         "TV: 315, \n" +
         "Refrigerator: 400";
      System.out.println(catalog);
      /* MicrowaveOven: 170,
         AudioSystem: 125,
         TV: 315,
         Refrigerator: 400 */

      String[] products = catalog.split("[\\d\\s,:]+");
      for(String product : products) {
         System.out.print(product + " ");
      }
      // MicrowaveOven AudioSystem TV Refrigerator
      System.out.println();

      String[] prices = catalog.split("\\D+");
      for(String price : prices) {
         System.out.print(price + " ");
      }
      // Result: 170 125 315 400
   }
}
```

Структурата на каталога е еднотипна: **име_на_продукт: цена**, последвани от нов ред. При първото разделяне използваме класа **\D**, като разделяме текста на всички символи, които не са цифри. След това, за да изведем единствено имената на продуктите, използваме по-сложно множество: **[\d\s,:]**, което обхваща всички цифри, всички интервали (необходимо ни е за премахването на разстоянията и знаците за нов ред), запетаи и двоеточия.

Виждате, че преди използваните класове е необходимо поставянето на още една наклонена черта, която указва, че наклонената черта от дефиницията на класа не е екраниращ символ.

Могат да бъдат дефинирани много по-сложни и прецизни регулярни изрази, които да решават по-специфични проблеми.

Повече информация за регулярните изрази и как да се възползваме от тях в Java може да откриете в уроците на Sun: java.sun.com/docs/books/tutorial/essential/regex/ или в онлайн Java API спецификацията: java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html.

Замяна на подниз с друг

При необходимост разполагаме и с готови методи за подмяна на един подниз с друг. Това може да се наложи, ако сме допуснали една и съща техническа грешка при въвеждане на email адреса на даден потребител в официален документ. Няма страшно – макар в целия документ адресът на потребителя да е сгрешен, може да го заменим с помощта на метода **replace(…)**:

```
String doc = "Hello, some@mail.bg, " +
   "you have been using some@mail.bg in your registration.";
String fixedDoc = doc.replace("some@mail.bg", "osama@laden.af");
System.out.println(fixedDoc);
```

Първоначалният ни текст съдържа предходния контакт на потребителя: **some@mail.bg**. След корекцията от наша страна и заместването на низа с метода **replace(…)**, всички предходни места, на които е срещан старият низ, са заменени със стойност **osama@laden.af**.

Реализиран е и еквивалентен, но универсален вариант, който замества поднизовете, отговарящи на даден регулярен израз. Когато се налага да работим с по-обща информация, на помощ ни идва метода **replaceAll(…)**.

Замяна на телефони със звездички – пример

Ако имаме служебен документ, който се използва само в офиса, и в него има лични данни, можем да ги цензурираме, преди да ги пратим на клиента. Например, има възможност да цензурираме всички номера на мобилни телефони и да ги заместим със звездички. Заданието е реализирано в следващия пример:

```
String doc = "Smith's number: 0892880022 \n"+
     "Franky can be found at 0853445566 \n" +
     "so as Steven - 0811654321";
replacedDoc = doc.replaceAll("(08)[0-9]{8}", "$1********");
System.out.println(replacedDoc);
```

Изходът на конзолата изглежда така:

```
Smith's number: 08**********
```

```
Franky can be found at 08**********
so as Steven - 08**********
```

Обяснение на аргументите на replaceAll(...)

В горния фрагмент от код използваме регулярен израз, с който откриваме всички телефонни номера в зададения ни текст. Променливата, която имитира документа с текстовите данни, е **doc**. В нея са записани, подобно на телефонен указател, няколко имена на клиенти, придружени от техните телефонни номера. Ако искаме да предпазим контактите от неправомерно използване и желаем да цензурираме телефонните номера, то може да заменим всички мобилни телефони със звездички. Приемайки, че телефоните са записани под формат: "**08 + 8 цифри**", методът **replaceAll(…)** открива всички съвпадения на дадения формат и ги замества с: "**08********"**. Регулярният израз, отговорен за откриването на номерата, е "**(08)[0-9]{8}**" – проста проверка за всички поднизове в текста, изградени от константата "**08**" и следвани от точно 8 символа в интервала от 0 до 9. Примерът може да бъде допълнително подобрен за подбиране на номерата само от дадени мобилни оператори, за работа с телефони на чуждестранни мрежи и др., но за целта на примера е използван опростен вариант. Литералът "**08**" е заграден от кръгли скоби. Те служат за обособяване на отделна група от регулярния израз. Групите могат да бъдат използвани за обработка само на определена част от израза, вместо целия. В нашия пример, групата е използвана в заместването – откритите съвпадения са заместени с "**$1********"** – текстът от първата група на регулярния израз + последователни 8 звездички за цензурата. Тъй като дефинираната от нас група винаги е константа – 08, то заместеният текст ще бъде във формат: **08********.**

Преминаване към главни и малки букви

Понякога имаме нужда да променим съдържанието на символен низ, така че всички символи в него да бъдат само с главни или малки букви. Двата метода, които биха ни свършили работа в случая, са **toLowerCase()** и **toUpperCase()**. Първата функция конвертира всички главни букви към малки:

```
String text = "All Kind OF LeTTeRs";
System.out.println(text.toLowerCase());
// all kind of letters
```

В примера се вижда, че всички главни букви от текста сменят регистъра си и целият текст остава изцяло с малки букви.

Ако искаме да сравним въведен вход от потребителя и не сме сигурни по какъв точно начин е написан той, можем да уеднаквим регистъра на буквите и да го сравним с дефинираната от нас константа. По този начин

не правим разлика за малки и главни букви. Например, ако имаме входен панел на потребителя, в който въвеждаме име и парола, и няма значение дали паролата е написана с малки, или главни букви, може да направим подобна проверка:

```
String pass1 = "Parola";
String pass2 = "PaRoLa";
String pass3 = "parola";
boolean isEqual;
isEqual = pass1.toUpperCase().equals("PAROLA") ; // true
isEqual = pass2.toUpperCase().equals("PAROLA") ; // true
isEqual = pass3.toUpperCase().equals("PAROLA") ; // true
```

В примера сравняваме 3 пароли с еднакво съдържание, но различен регистър, като при крайната проверка съдържанието им е еквивалентно на "PAROLA". В този случай малко обезсмисляме действието на метода **`equalsIgnoreCase(…)`**, като дефинираме проверката ръчно.

Премахване на празно пространство в края на низ

Въвеждайки текст във файл или през конзолата, понякога се появяват 'паразитни' интервали в началото или в края на текста. В началото или след края на дадена променлива може да се запише неволно някой друг интервал или табулация, които да не могат да се доловят на пръв поглед. Това може да не е съществено, но ако валидираме потребителски данни, би било проблем от гледна точка на проверка съдържанието на входната информация. На помощ идва методът **`trim()`** – той се грижи именно за премахването на паразитните празни места. Извиквайки метода на променлива от тип **`String`**, която има празни места в началото или края, той ще се погрижи за премахването им. Празните места могат да бъдат интервали, табулация, нови редове и др.

Ако в променливата **`fileData`** сме прочели съдържанието на файл, в който е записано име, а пишейки текста или преобръщайки го от един формат в друг са се появили паразитни интервали, променливата може да изглежда по подобен начин:

```
String fileData = "   \n\n     Mario Peshev      ";
```

Ако изведем съдържанието на конзолата, ще получим 2 празни реда, последвани от няколко интервала, търсеното от нас име и още няколко допълнителни интервала в края. Тъй като на нас ни е необходимо само името, може да редуцираме информацията от променливата и да премахнем ненужните интервали:

```
String reduced = fileData.trim();
```

Когато изведем повторно информацията на конзолата, съдържанието ще бъде "**Mario Peshev**", без нежеланите интервали.

Построяване на символни низове. StringBuilder

Както казахме по-горе, символните низове в Java са неизменими. Това означава, че всички корекции, приложени върху съществуващ низ, връщат като резултат нов символен низ. Например, използването на методите **replace(…)**, **toUpperCase()**, **trim()** не променят стринга, за който са извикани, а заделят нова област в паметта, в която се записва новото съдържание. Това има много предимства, но в някои случаи може да ни създаде проблеми с производителността на приложенията ни, ако не знаем тази съществена особеност.

Долепяне на низове в цикъл: никога не го правете!

Подобен проблем може да срещнем, когато се опитаме да съединяваме символни низове в цикъл, без значение от това дали конкатенацията е извършена чрез **concat(…)** метода или с операторите **+** и **+=**. Проблемът е пряко свързан с обработката на низовете и динамичната памет, в която се съхраняват те.

Как работи съединяването на низове?

Вече се запознахме с процедурата по съединяване на низове в Java. Да вземем за пример 2 дефинирани променливи **str1** и **str2** от тип **String**, които имат стойности съответно "Super" и "Star". В хийпа (динамичната памет) имаме две области, в които се съхраняват стойностите. Задачата на **str1** и **str2** е да пазят препратка към адресите в паметта, на които се намират записаните от нас данни. Нека създадем променлива **result** и ѝ придадем стойността на другите 2 низа. Фрагментът от код за създаването и дефинирането на трите променливи би изглеждал така:

```
String str1 = "Super";
String str2 = "Star";
String result = str1 + str2;
```

Създаването на променливата **result** ще задели нова област от динамичната памет, в която ще запише резултата от **str1 + str2**, който е "SuperStar". След това самата променлива ще пази адреса на заделената област. Като резултат ще имаме 3 области в паметта, както и 3 референции към тях. Това е удобно и ясно, но създаването на нова област, записването на стойност, създаването на нова променлива и реферирането ѝ към паметта е времеотнемащ процес, който би бил проблем при многократното му повтаряне в цикъл.

За разлика от други езици за програмиране, в Java не е дефинирано понятието "деструктор", т.е. не е необходимо ръчното освобождаване на

обектите, записани в паметта. Съществува специален механизъм, наречен **garbage collector (система за почистване на паметта)**, който се грижи за изчистването на неизползваната памет и ресурси. Системата за почистване на паметта е отговорна за освобождаването на динамичната памет, когато вече не се използва. Създаването на много обекти, придружени с множество референции в паметта, е вредно, защото запълват паметта и се налага автоматичното изпълнение на garbage collector от виртуалната машина. Това отнема немалко време и забавя цялостното изпълнение на процеса.

Защо долепянето на низове в цикъл е лоша практика?

Да приемем, че имаме за задача да запишем числата от 1 до 5000 в променлива от тип **`String`**. Как можем да решим задачата с досегашните си знания? Един от най-лесните начини за имплементация е създаването на променливата, която съхранява числата, и завъртането на цикъл от 1 до 5000, в който всяко число се долепва към въпросната променлива. Реализирано на Java решението би изглеждало така:

```
String collector = "Numbers: ";
for (int idx = 1; idx <= 5000; idx++) {
   collector += idx;
}
```

Изпълнението на горния код ще завърти цикъла 5000 пъти, като след всяко завъртане ще добавя текущия индекс към променливата **`collector`**. Стойността на променливата след края на изпълнението ще бъде: "Numbers: 1234567891011121314151 6..." (с многоточие са заместени останалите числа от 17 до 5000 с цел относителна представа за резултата).

Вероятно не ви е направило впечатление забавянето при изпълнение на фрагмента. Всъщност използването на конкатенацията в цикъл е забавила значително нормалното развитие на процеса и на средностатистически процесор (към август 2008 г.) итерацията на цикъла отнема 2-4 секунди. Ползвателят на програмата ни би бил доста скептично настроен, ако се налага да чака няколко секунди за нещо елементарно, като слепване на числата от 1 до 5000. Освен това в случая 5000 е само примерна крайна точка. Какво ли ще бъде забавянето, ако вместо 5000, потребителят има нужда от числата до 50000? Пробвайте!

Конкатениране в цикъл с 50000 итерации - пример

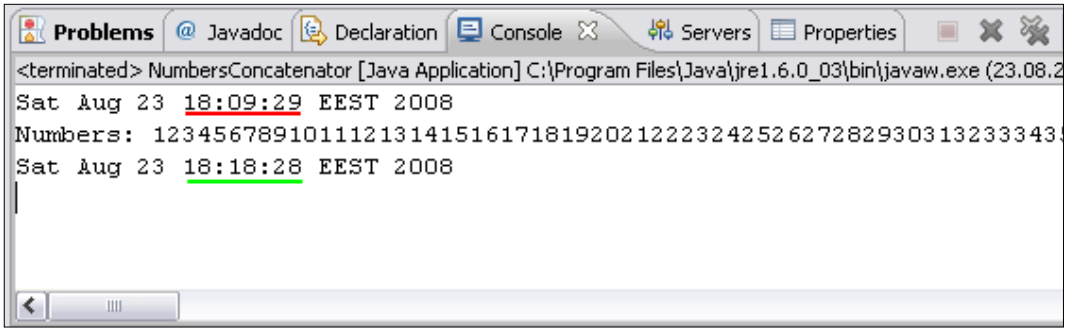
Нека да развием горния пример. Първо, ще променим крайната точка на цикъла от 5000 на 50000. Второ, за да отчетем правилно времето за изпълнение, ще извеждаме на конзолата текущата дата и час преди и след изпълнението на цикъла. Трето, за да видим, че променливата съдържа желаната от нас стойност, ще изведем част от нея на конзолата. Причината да не извеждаме цялата променлива е, че конзолата има буфер с определен размер и при стандартни настройки не може да изведе пълната на

променливата. Ако искате да се уверите, че цялата стойност е запаметена, може да увеличите ръчно размера на буфера от настройките на конзолата на Eclipse (**Window | Preferences | Run/Debug | Console**) или да запишете съдържанието на променливата в текстов файл.

Крайният вариант на примера би изглеждал така:

```
import java.util.Date;

public class NumbersConcatenator {
   public static void main(String[] args) {
      System.out.println(new Date());

      String collector = "Numbers: ";
      for(int idx = 1; idx <= 50000; idx++) {
         collector += idx;
      }

      System.out.println(collector.substring(0, 1024));
      System.out.println(new Date());
   }
}
```

При изпълнението на примера в конзолата се извеждат дата и час на стартиране на програмата, отрязък от първите 1024 символа от променливата, както и дата и час на завършване на програмата. Причината да отрежем първите 1024 символа е, че самото отпечатване на голям обем текстова информация на конзолата отнема доста време, а ние искаме да измерим само времето за изчисленията без времето за отпечатване на резултата. Нека видим примерния изход от изпълнението:



С червена линия е подчертан таймерът в началото на програмата, а със зелена – нейният край. Обърнете внимание на времето за изпълнение – почти 9 минути! Представете си, ако при стартиране на операционната система трябва да чакате 9 минути, за да получите съобщение за приветствие и текущата дата? Подобно изчакване е недопустимо за подобна задача.

Обработка на символни низове в паметта

Проблемът с дълготрайната обработка на цикъла е свързан именно с работата на стринговете в паметта. Всяка една итерация създава нов обект в динамичната памет и насочва референцията към него. Процесът изисква определено физическо време.

На всяка стъпка се случват няколко неща:

1. Заделя се област от паметта за записване на резултата от долепването на поредната цифра. Тази памет се използва само временно, докато се изпълнява долепването, и се нарича **буфер**.
2. Премества се старият низ в ново заделения буфер. Ако низът е дълъг (примерно 1 MB или 10 MB), това може да е доста бавна операция!
3. Долепя се поредната цифра към буфера.
4. Буферът се преобразува в символен низ.
5. Старият низ, както и временният буфер, остават неизползвани и по някое време биват унищожени от системата за почистване на паметта (**garbage collector**). Това също може да е бавна операция.

Много по-елегантен и удачен начин за конкатениране на низове в цикъл е използването на класа **`StringBuilder`**.

Построяване и промяна на низове със StringBuilder

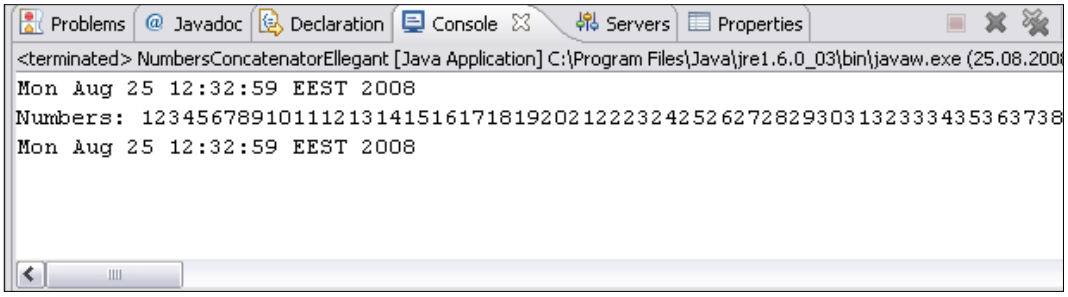
`java.lang.StringBuilder` е клас, който служи за построяване и промяна на символни низове. Той преодолява проблемите с бързодействието, които възникват при конкатениране на низове от тип **`String`**. Класът е изграден под формата на масив от символи и това, което трябва да знаем за него, е че информацията в него не е неизменима – промените, които се налагат в променливите от тип **`StringBuilder`**, се извършват в една и съща област от паметта (буфер), което спестява време и ресурси. За промяната на съдържанието не се създава нов обект, а просто се променя текущият.

Нека сравним горния пример, в който слепвахме низове в цикъл, като операцията ни отне 9 минути. Много по-елегантно решение е използването на **`StringBuilder`** за подобен род задачи. Нека видим алтернативно решение на същата задача:

```
import java.util.Date;

public class NumbersConcatenatorEllegant {
   public static void main(String[] args) {
      System.out.println(new Date());

      StringBuilder sb = new StringBuilder();
      sb.append("Numbers: ");
```

```
      for(int idx = 1; idx <= 50000; idx++) {
         sb.append(idx);
      }

      System.out.println(sb.substring(0, 1024));
      System.out.println(new Date());
   }
}
```

Примерът е базиран на предходния, със съвсем леки корекции. Връщаният резултат е същият, а какво ще кажете за времето за изпълнение?



Необходимото време за слепване на 50000 символа със **StringBuilder** е по-малко от секунда!

Обръщане на низ на обратно – пример

Да разгледаме пример, в който искаме да обърнем съществуващ символен низ на обратно. Например, ако имаме низа "ABCD", върнатият резултат ще бъде "DCBA". Това, което ще направим, е да вземем първоначалния низ, да го обходим отзад-напред символ по символ и да добавяме всеки символ към променлива от тип **StringBuilder**:

```
public class WordReverser {
   public static void main(String[] args) {
      String text = "EM edit";
      String reversed = reverseText(text);
      System.out.println(reversed); // tide ME
   }

   public static String reverseText(String text) {
      StringBuilder sb = new StringBuilder();
      for (int i = text.length() - 1; i >= 0; i--)
         sb.append(text.charAt(i));
      return sb.toString();
   }
}
```

В демонстрацията имаме променливата **text**, която има стойност "EM edit". Подаваме променливата на метода **reverseText(…)** и приемаме новата стойност в променлива с име **reversed**. Методът, от своя страна, обхожда символите от променливата в обратен ред и записва символите в нова променлива от тип **StringBuilder**, но вече наредени обратно. В крайна сметка резултатът е "tide ME".

Как работи класът StringBuilder?

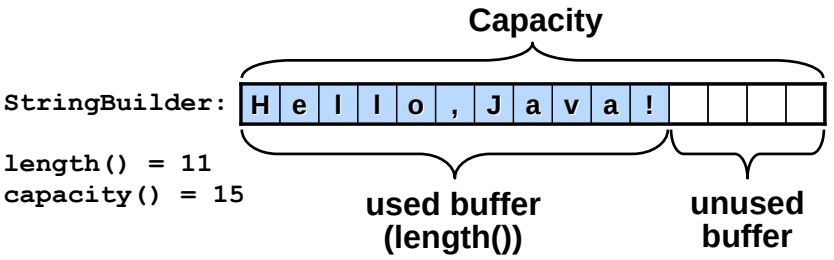
Класът **StringBuilder** представлява реализация на символен низ в Java, но различна от тази на класа **String**. За разлика от познатите вече символни низове, обектите на класа **StringBuilder** не са неизменими, т.е. редакциите не налагат създаването на нов обект в паметта.

StringBuilder поддържа буфер с определен капацитет (по подразбиране 16 символа). Буферът е реализиран под формата на масив от символи, който е предоставен на програмиста с удобен интерфейс – методи за лесно и бързо добавяне, търсене, редактиране на елементите на стринга. Във всеки един момент част от символите в буфера се използват, а останалите стоят в резерва. Това дава възможност добавянето да работи изключително бързо. Останалите операции също работят по-бързо, отколкото при класа **String**, защото промените не създават нов обект.

Нека създадем обект от класа **StringBuilder** с буфер от 15 символа. Към него ще добавим символния низ: "Hello,Java!". Реализирано с Java, заданието ни ще изглежда така:

```
StringBuilder sb = new StringBuilder(15);
sb.append("Hello,Java!");
```

След създаването на обекта и записването на стойността в него, той би изглеждал по подобен начин:



Оцветените елементи са запълнената част от буфера с въведеното от нас съдържание. Обикновено при добавяне на нов символ към променливата не се създава нов обект в паметта, а се използват заетото пространство за редакции и неизползваното за добавяне на нови данни. Ако целият капацитет на буфера е запълнен, тогава вече се заделя нова област в динамичната памет с удвоен размер (текущия капацитет + 1, умножен по

2). След това можем отново да добавяме спокойно символи и символни низове, без да се притесняваме за непрекъснатото заделяне на памет.

StringBuilder – по-важни методи

Класът **StringBuilder** ни предоставя набор от методи, които ни помагат за лесна и ефективна работа с променливите. Някои от тях са:

- **StringBuilder(int capacity)** – конструктор с параметър начален капацитет. Чрез него може предварително да зададем размера на буфера, ако имаме приблизителна информация за броя итерации и слепвания. Така спестяваме излишни заделяния на динамична памет.
- **capacity()** – връща размера на целия буфер (общия брой заети и свободни символи)
- **length()** – връща дължината на записания низ в променливата
- **charAt(int index)** – връща символа на указаната позиция
- **append(…)** – слепва низ, число или друга стойност след последния записан символ в буфера
- **delete(int start, int end)** – премахва низ по зададена начална и крайна позиция
- **insert(int offset, String str)** – вмъква даден стринг на дадена позиция
- **replace(int start, int end, String str)** – замества записания низ между началната и крайната позиция със стойността на променливата str
- **toString()** – връща записаната информация в обекта на **StringBuilder** като резултат от тип **String**, който можем да запишем в променлива на String.

Извличане на главните букви от текст – пример

Следващата задача е да извлечем всички главни букви от един текст. Можем да я реализираме по различни начини – използвайки масив и пълнейки масива с всички открити главни букви; създавайки обект от тип **String** и долепвайки главните букви към него; използвайки класа **StringBuilder**.

Спирайки се на варианта за използване с масив, ние имаме един конкретен проблем: не знаем какъв да бъде размерът на масива, тъй като предварително нямаме идея колко са главните букви в текста. Така че се опасяваме дали масивът ще бъде достатъчно голям, за да побере необходимата ни информация. Може да създадем и масив с огромен размер, но по този начин хабим излишно място.

Друг вариант е използването на променлива от тип **String**. Тъй като ще обходим целия текст и ще долепваме всички букви към променливата, вероятно е отново да загубим производителност от гледна точка на конкатенирането на символни низове.

StringBuilder – правилното решение в случая

Най-уместното решение за задачата ни е използването на **StringBuilder**. За да решим коректно проблема от условието, ние създаваме променлива от класа, итерираме зададения текст символ по символ, проверяваме дали текущият символ от итерацията е главна буква и при положителен резултат долепваме символа към нашия обект. Накрая връщаме четим резултат с извикването на метода **toString()**.

Реализацията на алгоритъма с Java можем да открием в следния фрагмент:

```
public static String extractCapitals(String s) {
   StringBuilder result = new StringBuilder();
   for (int i = 0; i < s.length(); i++) {
     char ch = s.charAt(i);
     if (Character.isUpperCase(ch)) {
        result.append(ch);
      }
   }
   return result.toString();
}
```

Извиквайки метода **extractCapitals(…)** и подавайки му зададен текст като параметър, връщаната стойност е низ от всички главни букви в текста. За проверка на главните букви използваме **Character. isUpperCase(…)** – готов метод в от стандартните класове в Java, който проверява дали даден символ е главна буква или не, като връща булев резултат.

Можете да разгледате документацията за класа **Character**, защото той предлага и други полезни методи за обработка на символи.

Форматиране на низове

Java предлага на програмиста механизми за форматиране на символните низове. Практически всеки създаден обект на клас, както и примитивните променливи, могат да бъдат представени като текстово съдържание. Налице са форматиращи класове и методи, които служат за правилното форматиране на текст, числа, дати. Спомнете си метода **printf(…)** от **System.out.printf(…)** – с негова помощ извеждаме символни низове с предварително форматирано съдържание, можем да задаваме шаблони, в които да попълваме празните места с променливи или литерали; да форматираме дати, числа и т.н.

С някои от средствата за форматиране на текст вече се запознахме в главата "Вход и изход от конзолата". Ще преговорим по-важните от тях и ще допълним знанията си за форматирането и преобразуването на низове.

Класът java.util.Formatter

java.util.Formatter дава възможност за извеждане на форматиращи символни низове. Сред възможностите на класа са подравняването на текста и различни методи за форматиране на текст, символи, дати и специфичен изход в зависимост от местоположението. Създаването на класа е вдъхновено от функцията **printf(…)** в езика C, като имплементацията е реализирана със сходен синтаксис, но с по-стриктни изисквания, съобразени с езика Java.

Всеки метод, който връща форматиран изход, изисква форматиращ стринг и списък от аргументи. Форматиращият низ е **String** обект, който съдържа фиксиран текст и един или повече вложени форматиращи спецификатори (**format specifiers**). Основните спецификатори за символни и числови типове имат следния синтаксис:

```
%[индекс_на_аргумента$][флагове][ширина][.точност]формат
```

- **индекс_на_аргумента** – незадължителен спецификатор; десетично число, указващо позицията на аргумента. Първият аргумент има индекс "1$", вторият – "2$", и т.н.
- **флагове** – незадължителен спецификатор; списък от символи, модифициращи начина на извеждане на низа. Зависи пряко от формата.
- **ширина** – незадължителен спецификатор; неотрицателно десетично число, посочващо минималния брой от символи, които да бъдат изведени на изхода. Удобен за таблично форматиране.
- **точност** – незадължителен спецификатор; неотрицателно десетично число, ограничаващ броя символи. Зависи от типа формат, широко използван при десетични числа.
- **формат** (**conversion**) – символ, указващ как да бъде форматиран аргументът. Зависи от типа на подадения аргумент.

Служебният метод toString()

Един от основните ни помощници за представянето на обектите като символни низове е методът **toString()**. Той е заложен в дефиницията на класа **Object** – базовият клас, който наследяват пряко или не всички референтни типове в езика. По този начин дефиницията на методи се появява във всеки един клас, като ние имаме възможност да изведем под някаква форма съдържанието на един обект като текст.

Методът **toString()** се извиква автоматично, когато извеждаме на конзолата обекти на различни класове. Например, когато боравим с дати, ние извеждаме текущата дата по следния начин:

```
Date currentDate = new Date();
System.out.println(currentDate);
// Thu Aug 28 11:34:27 EEST 2008
```

Когато подаваме **currentDate** като параметър на метода **println(…)**, нямаме точна декларация, която обработва дати. Методът има конкретна реализация за всички примитивни типове и символни низове. За всички останали обекти **println(…)** извиква метода **toString()**, който извежда съдържанието на обекта. Т.е. горният код е еквивалентен на този:

```
Date currentDate = new Date();
System.out.println(currentDate.toString());
```

Имплементацията на метода по подразбиране в класа **Object** връща уникална стойност на обекта, като извежда пълния път до класа и неговия хеш код. Всички класове, които не предефинират поведението на **toString()**, използват именно тази имплементация. Повечето класове в Java API имат предефинирана стойност на метода, представяща четимо и разбираемо съдържание на обектите. Препоръчително е предефинирането на метода в класовете, създавани от програмиста.

Използване на String.format()

String.format() е статичен метод, чрез който можем да създаваме форматирани стрингове, на които да подаваме параметри. Той е удобен при създаването на шаблони – често срещани текстове с променливи параметри. С негова помощ можем да използваме низове с декларирани параметри и всеки път да променяме единствено параметрите. Можем да направим асоциация с метода **System.out.printf(…)**, който също форматира шаблонен низ и подава стойности на местата на параметрите:

```
System.out.printf("This is a template from %s.", "Mario");
```

Както методът **String.format()**, така и **System.out.printf(…)** ползват за извеждането на параметризираните низове класа **java.util.Formatter**.

Преобразуване на типове

Полезно свойство на Java е възможността за преобразуване на типове – преминаването на променлива от един тип в друг. Често работата с приложения с графичен потребителски интерфейс предполага потребителският вход да бъде предаван през променливи от тип String, защото практически така може да се работи както с числа и символи, така и с текст и дати, форматирани по предпочитан от нас начин. Въпрос на опит на

програмиста е да представи входните данни, които очаква, по правилния начин на потребителя, за да получи подходящи входни данни. След това данните се преобразуват към по-конкретен тип и се обработват. Например числата могат да се преобразуват като променливи от **int** или **double**, а след това да участват в математически изрази за изчисления.

При преобразуването на типове не бива да се осланяме само на доверието към потребителя. Винаги проверявайте входните потребителски данни при преобразуване! В противен случай ще настъпи изключение.

Преобразуване към числови типове

За преобразуване на символен низ към число можем да използваме **обвиващите класове (wrapper classes)** на примитивните типове. По-горе в темата използвахме един от тези класове, а именно **Character**. Всички примитивни типове имат прилежащите им класове, които служат за представянето на примитивна стойност като обект и предоставят често използвани методи, които можем да ползваме наготово. Например типът **char** предлага клас **Character**, типът **float** – клас **Float**, типът **int** - клас **Integer**, и т.н.

Обвиващите класове, като изключим **Character**, предлагат методи за преобразуването на текстова променлива към променлива от примитивния тип, с който е обвързан с обвиващия клас. Методите имат формата **parseXXX(String)**, като на мястото на **XXX** е името на типа (например **parseInt(…)**, **parseBoolean(…)** и др.). Нека видим пример за преобразуване на целочислена стойност (парсване):

```
String text = "53";
int intValue = Integer.parseInt(text); // 53
```

Можем да преобразуваме и променливи от булев тип:

```
String text = "True";
Boolean boolValue = Boolean.parseBoolean(text); // true
```

Връщаната стойност е **true**, когато подаваният параметър е инициализиран (не е обект със стойност **null**) и съдържанието й е "**true**", без значение от малките и главни букви в него, т.е. всякакви текстове като "true", "True" или "tRUe" ще зададат на променливата **boolValue** стойност **true**. Всички останали случаи връщат стойност **false**.

В случай, че подадената на **parseXXX** метод стойност е невалидна (например подаваме "Пешо" при парсване на число), се получава изключение **NumberFormatException**.

Обработване на дати – SimpleDateFormat

Датите са по-специфичен тип, който няма примитивно представяне, но тъй като са често използвани, ние имаме способи за тяхното преобразуване от текстов тип. Можем ли със сегашните ни знания да създадем дата от текстова променлива? Отговорът на този въпрос е "да". Например, ако форматът е "27.10.2008", можем да използваме метода **split(…)**, за да разделим съдържанието по точка; след това да обходим масива от връщани стойности, да ги преобразуваме като цели числа с **parseInt(…)** на класа Integer и да създадем нов календар, от който да вземем датата. С код нещата биха изглеждали така:

```
String text = "27.10.2008";
String[] dateElements = text.split("[.]");
String dayString = dateElements[0];
String monthString = dateElements[1];
String yearString = dateElements[2];

int day = Integer.parseInt(dayString);
int month = Integer.parseInt(monthString);
int year = Integer.parseInt(yearString);

Calendar cal = new GregorianCalendar(year, month - 1, day);
Date date = cal.getTime(); // Mon Oct 27 00:00:00 EET 2008
```

Виждаме, че в крайния резултат имаме променлива от тип **Date**, която е изградена чрез посочените от нас ден, месец и година. Ако обърнете внимание, в предпоследния ред месецът е подаден като **month – 1**. Причината за това е, че в Java месеците започват от 0 (т.е. януари е 0, февруари – 1, и т.н.), а в познатото от нас означаване месеците започват от 1. На календара на нашия компютър или мобилен телефон комбинацията 27.10 отбелязва месец октомври, докато в Java обект това би означавало двадесет и седми ноември.

Въпреки тази особено, изписахме доста код чрез доста ръчни проверки и преобразувания, за да стигнем до желания резултат. Класът **java.text.SimpleDateFormat** съдържа функционалност, чрез която достигаме до по-елегантно преобразуване на типовете. Той ни дава възможност за преобразуване на текстово съдържание към дата, както и обратното.

Не забравяйте да вмъкнете `java.text.SimpleDateFormat` или пакета `java.text.*` в началото на програмата, за да може да използвате възможностите на класа. Класът `Date` също се намира в `java.util` пакета и не се импортира автоматично!

java.text.SimpleDateFormat – шаблони

SimpleDateFormat ни предлага по-удобен интерфейс за превръщане на текстови променливи към обекти на класа **java.util.Date**, както и обратното действие. Той е базиран на работата на шаблони, които дефинират по какъв начин е въведена датата: кои елементи от нея са зададени (ден, месец, година, час, милисекунди и т.н.), какви са разделителите (точки, наклонени черти, интервали), използва ли се 24-часово визуализиране или 12-часово и други подробности. Ако потребителят е наясно с използвания формат, той може да въведе датата по дефинирания начин и тя ще бъде преобразувана в типа **Date**. Възможно е да предложим на потребителя и сам да избере типа на въвеждане на датата според шаблона.

Форматиращ символ	Описание
G	Ера
Y	Година
M	Месец
D	Ден от месеца
H	Час (1-12, сутрин/следобед)
H	Час (0-23)
K	Час (1-24)
K	Час (0-11, сутрин/следобед)
M	Минути
S	Секунди
S	Милисекунди (0-999)
E	Ден от седмицата
D	Ден от годината (1-365)
F	Ден от седмицата в месеца (1-5)
w	Седмица в годината (1-53)
W	Седмица в месеца (1-5)
A	Am/Pm – сутрин/следобед
Z	Времева зона

Ще разгледаме някои примери за различни комбинации с шаблоните за форматиране на дати. Отправната ни дата, преди форматиране, изглежда по следния начин:

```
Mon Oct 13 14:02:03 EEST 2008
```

Обърнете внимание, че можем да вмъкваме произволен текст във форматиращия низ с цел по-добра четимост.

Шаблон	Резултат
yyyy.MM.dd G 'at' HH:mm:ss z	2008.10.13 н.е. at 14:02:03 EEST
EEE, MMM d, ''yy	Пн, X 13, '08
H:mm a	2:02 PM
hh 'o''clock' a, zzzz	02 o'clock PM, Eastern European Summer Time
K:mm a, z	2:02 PM, EEST
yyyyy.MMMMM.dd GGG hh:mm aaa	02008.Октомври.13 н.е. 02:02 PM
EEE, d MMM yyyy HH:mm:ss Z	Пн, 13 X 2008 14:02:03 +0300
yyMMddHHmmssZ	081013140203+0300

Преобразуване на низове към дати – пример

Нека видим предходния пример за преобразуване на символен низ към дата, но този път да използваме класа **SimpleDateFormat** за форматиране. Ще дефинираме същият шаблон за дата: **ден.месец.година**:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy");
Date date = sdf.parse("27.10.2008");
System.out.println(date); // Mon Oct 27 00:00:00 EET 2008
```

Както виждате, използването на помощния клас **java.text.SimpleDateFormat** ни спестява доста излишно писане на код и ръчно преобразуване на типовете. Има и още една особеност: в предходния пример бе необходимо да намалим с единица стойността на променливата за месеца, тъй като в Java месеците са номерирани от 0 до 11. Методът **parse(…)** автоматично преобразува стойностите, преди да ги присвои на променлива от тип **Date**, и в текущия фрагмент месецът с номер 10 е октомври, а не ноември.

Важно е да се отбележи, че методът **parse(…)** предизвиква **java.text.ParseException**. Задължително е да прихванем това изключение по някакъв начин (било то в **try/catch** блок или чрез **throws** декларация на метода). Изключение реално настъпва, ако потребителският вход не отговаря на шаблона и въведеният символен низ не може да бъде преобразуван към дата – например промяна на разделителя от точка на наклонена черта, добавяне на 4-ти параметър, който не ни е познат; дефиниране на грешен шаблон и т.н.

Ето един примерен начин за прихващане на изключения при работата с метода **parse(…)**:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy");
String userInput = "27/12/2008";
try {
   Date date = sdf.parse(userInput);
   System.out.println(date);
}
catch(ParseException pe) {
   System.out.println("Error in parsing " + userInput);
   // Error in parsing 27/12/2008
}
```

Ако входът от потребителя е коректен и съвпада с шаблона, то нашият код ще бъде изпълнен успешно. В посочения пример данните са разделени от наклонена черта, докато шаблонът ни очаква за разделител точка. В такъв случай ще бъде хвърлено изключение, което ние извеждаме на конзолата със съобщение за грешка. Във всички случаи е необходимо прихващане на изключенията при конверсията на потребителски вход към дата.

Въвеждането на грешна стойност за ден или месец може да не генерира изключение! В Java е допустимо създаването на променлива 'ден' със стойност по-голяма от 31 или 'месец', чиято стойност надхвърля 12. Например при задаването на месец с по-голяма стойност от 12, ще преминем в следващата година.

Горната забележка е сходна с 'прехвърляне на брояча' на хронометър – когато секундите надхвърлят 59, новата стойност става 0, а минутите се увеличават с единица (аналогично за часове, дни, месеци и т.н). В използвания от нас календар месеците са от 1 до 12 – респективно от януари до декември, но в Java е възможно да зададем месец 13, без да получим изключение. Задаването на 27.13.2008 г. например е еквивалентно на 27.01.2009 г.:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy");
Date date = sdf.parse("27.13.2008");
System.out.println(date); // Tue Jan 27 00:00:00 EET 2009
```

Въпреки това е препоръчително да не се използват такива стойности, когато е възможно, за да не настъпва объркване.

Преобразуване на дати към символни низове – пример

Още един полезен метод на класа **SimpleDateFormat** е **format(…)**. С негова помощ можем да преобразуваме дати към символни низове. Вече знаем, че обектите на класа **Date** могат да бъдат извеждани на конзолата. Често се

налага да извеждаме датата, форматирана по желан от нас начин. Стандартното форматиране визуализира деня от седмицата, текущия месец, деня от месеца, час, минути, секунди, часова зона и година. В частни случаи се нуждаем само от определена част от датата – например само часовете и минутите. Тогава създаваме шаблон, който да извежда часовете и минутите от датата, като ни спестява извеждането на другата информация.

Нека създадем обект от тип **Date**, който съдържа текущата дата, и да изведем часа и минутите, в които е създаден обекта:

```
String pattern = "HH часа и mm мин.";
SimpleDateFormat sdf = new SimpleDateFormat(pattern);
Date dateNow = new Date();
System.out.println(dateNow); // Sun Oct 12 15:57:39 EEST 2008
String formattedDate = sdf.format(dateNow);
System.out.println(formattedDate); // 15 часа и 57 мин.
```

Методът **format(…)** приема като аргумент обект от тип **Date** и връща стойността от тип **String**, която можем да използваме, за да я покажем на потребителя или да я обработим по някакъв начин. За разлика от **parse(…)**, **format(…)** не хвърля винаги изключение, т.е. не е необходимо всеки път да прихващаме евентуален **ParseException**. При некоректен шаблон обаче може да възникне **IllegalArgumentException**. В стандартния случай шаблоните и създадените дати са създадени от програмиста и няма опасност от възникване на изключение.

Няма проблем за добавяне на допълнителни символи за по-прегледен вид на датата. В примера по-горе сме добавили думи "часа" и "мин." с цел по-голяма яснота за потребителите. Шаблоните "HH" и "mm" пък са заместител на часа и минутите от обекта **dateNow**.

За финал, ето и един пример за извеждане на деня от седмицата и текущата дата (елементите ден, месец, година):

```
String pattern = "EEEE, dd.MM.yyyy г.";
SimpleDateFormat sdf = new SimpleDateFormat(pattern);
Date dateNow = new Date();
System.out.println(dateNow); // Sun Oct 12 16:09:04 EEST 2008
String formattedDate = sdf.format(dateNow);
System.out.println(formattedDate); // Неделя, 12.10.2008 г.
```

Упражнения

1. Напишете програма, която прочита символен низ, обръща го отзад напред и го принтира обратно на конзолата. Например: "introduction" → "noitcudortni".

2. Напишете програма, която открива колко пъти даден подниз се съдържа в текст. Например, ако търсим подниза "in" в текста:

```
We are living in a yellow submarine. We don't have anything else.
Inside the submarine is very tight. So we are drinking all the
day. We will move out of it in 5 days.
```

   Резултатът е 9.

3. Даден е текст. Напишете програма, която променя регистъра на буквите на всички места в текста, заградени с таговете **<upcase>** и **</upcase>**. Таговете не могат да бъдат вложени.

   Пример:

```
We are living in a <upcase>yellow submarine</upcase>. We don't
have <upcase>anything</upcase> else.
```

   Резултат:

```
We are living in a YELLOW SUBMARINE. We don't have ANYTHING else.
```

4. Даден е символен низ, съставен от няколко "забранени" думи, разделени със запетая. Даден е и текст, съдържащ тези думи. Да се напише програма, която замества забранените думи в текста със звездички. Пример:

```
Microsoft announced its next generation Java compiler today. It
uses advanced parser and special optimizer for the Microsoft JVM.
```

   Низ от забранените думи: "**Java,JVM,Microsoft**".

   Резултат:

```
********* announced its next generation **** compiler today. It
uses advanced parser and special optimizer for the ********* ***.
```

5. Напишете програма, която приема URL адрес във формат:

```
[protocol]://[server]/[resource]
```

   и извлича от него протокол, сървър и ресурс. Например, при подаден адрес: **http://www.devbg.org/forum/index.php** резултатът е:

   [protocol]="http"

   [server]="www.devbg.org"

   [resource]="/forum/index.php"

6. Напишете програма, която обръща думите в дадено изречение. Например: "C# is not C++ and PHP is not Delphi" -> "Delphi not is PHP and C++ not is C#".
7. Колко обратни наклонени черти трябва да посочите като аргумент на метода **`split(…)`**, за да разделите текста по обратна наклонена черта?

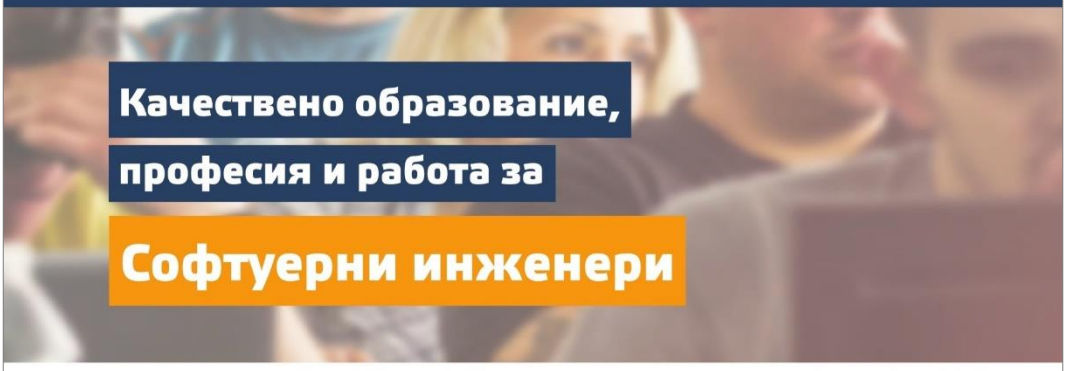
   Пример: **`one\two\three`**

   Забележка: В Java обратната наклонена черта е екраниращ символ (escaping character).

Решения и упътвания

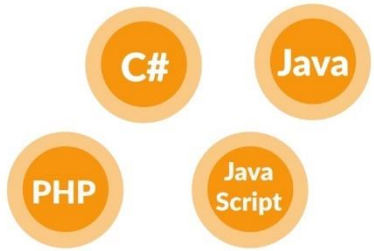
1. Използвайте **`StringBuilder`** и **`for`** цикъл.
2. Обърнете регистъра на буквите на текста и използвайте търсене в цикъл. Не забравяйте да използвате **`indexOf(…)`** с начален индекс, за да избегнете безкраен цикъл!
3. Използвайте регулярни изрази или **`indexOf(…)`** за отварящ и затварящ таг. Пресметнете началния и крайния индекс на текста. Обърнете текста в главни букви и заменете целия подниз **отварящ таг + текст + затварящ таг** с увеличения текст.
4. Разделете забранените думи с метода **`split(…)`**. За всяка забранена дума обхождайте текста и търсете срещане. При срещане на забранена дума, заменете с толкова звездички, колкото букви се съдържат в забранената дума. Броят може да установите с метода **`length()`**.
5. Използвайте регулярен израз или търсете по съответните разделители – две наклонени черти за край на протокол и една наклонена черта за разделител между сървър и ресурс.
6. Можете да решите задачата на две стъпки: обръщане на входния низ на обратно; обръщане на всяка от думите от резултата на обратно.
7. Ползвайте 4 наклонени черти: **`split("\\\\")`**.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 14. Дефиниране на класове

Автор

Николай Василев

Посвещение

Посвещавам тази глава, на първият ми учител по програмиране, доц. Божидар Сендов, от когото за пръв път видях как, за да може знанието да бъде разбрано от аудиторията, освен, че лекторът трябва да дава всичко от себе си в процеса на преподаване, той трябва да спечели умовете и сърцата на слушателите.

В тази тема...

В настоящата тема ще разберем как можем да дефинираме собствени класове и кои са елементите на класовете. Ще се научим да декларираме полета, конструктори и свойства в класовете. Ще припомним какво е метод и ще разширим знанията си за модификатори и нива на достъп до полетата и методите на класовете. Ще разгледаме особеностите на конструкторите и подробно ще обясним как обектите се съхраняват в динамичната памет и как се инициализират полетата им. Накрая ще обясним какво представляват статичните елементи на класа – полета (включително константи), свойства и методи и как да ги ползваме.

Собствени класове

"*... Всеки модел представя някакъв аспект от реалността или някаква интересна идея. Моделът е опростяване. Той интерпретира реалността, като се фокусира върху аспектите от нея, свързани с решаването на проблема и игнорира излишните детайли.*" [Evans]

Целта на всяка една програма, която създаваме, е да реши даден проблем или да реализира някаква идея. За да измислим решението, ние първо създаваме опростен модел на реалността, който не отразява всички факти от нея, а се фокусира само върху тези, които имат значение за намирането на решение на нашата задача. След това, използвайки модела, намираме решение (т.е. създаваме алгоритъма) на нашия проблем и това решение го описваме чрез средствата на даден език за програмиране.

В наши дни, най-често използваният тип езици за програмиране са обектно-ориентираните. И тъй като обектно-ориентираното програмиране (ООП) е близко до начина на мислене на човека, то ни дава възможността, с лекота да описваме модели на заобикалящата ни среда. Една от причините за това е, че ООП ни предоставя средство, за описание на съвкупността от понятия, които описват обектите във всеки модел. Това средство се нарича клас (class). Понятието клас и дефинирането на собствени класове, различни от системните, е част от езика Java и целта на тази глава е да се запознаем с него.

Да си припомним: какво са класовете и обектите?

Клас (**class**) наричаме описание на даден обект от реалността. Класът представлява шаблон, който описва видовете състояния и поведението на обектите (екземплярите), които биват създавани от този клас (шаблон).

Обект (**object**) наричаме екземпляр създаден по дефиницията (описанието) на даден клас. Когато един обект е създаден по описанието, което един клас дефинира, казваме, че **обектът е от тип "името на този клас"**.

Например, ако имаме клас **`Dog`**, описващ някакви характеристики на куче от реалния свят, казваме, че обектите, които са създадени по описанието на този клас са от тип – класът **`Dog`**. Това означение е същото, като например, низът **`"some string"`** казваме, че е от тип **`String`**. Разликата е, че обектът от тип **`Dog`** е екземпляр от клас, който не е част от библиотеката с класове на Java, а е дефиниран от самите нас.

Какво съдържа един клас?

Класът съдържа дефиниция на това какви данни трябва да се съдържат в един обект, за да се опише състоянието му. Обектът (конкретния екземпляр от този клас) съдържа самите данни. Тези данни дефинират състоянието му.

Освен състоянието, в класа също се описва и поведението на обектите. Поведението се изразява в действията, които могат да бъдат извършвани от обектите. Средството на ООП, чрез което можем да описваме поведението на обектите от даден клас, е деклариране на методи в класа.

Елементи на класа

Сега ще изброим основните елементи на един клас, а по-късно ще разгледаме подробно всеки един от тях.

Основните елементи на класа са следните:

- **Декларация на класа (class declaration)** – това е редът, на който декларираме името на класа. Например:

```
public class Dog {
```

- **Тяло на клас** – по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "**{**" и "**}**", между които се намира съдържанието на класа. Тя се нарича тяло на класа. Елементите на класа, които се описват в тялото му са изброени в следващите точки.

```
public class Dog {
   // ... Here the class body comes ...
}
```

- **Конструктор (constructor)** – това е псевдометод, който се използва за създаване на нови обекти. Така изглежда един конструктор:

```
public Dog() {
   // ... Some code ...
}
```

- **Полета (fields)** – полетата са променливи (някъде в литературата се срещат като **член-променливи**), декларирани в класа. В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени статични, които са общи за всички обекти.

```
// Field/Property-storage definition
private String name;
```

- **Свойства (properties)** – наричаме характерните особености на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект, или да са споделени между всички обекти от тип даден клас.

```
// Field/Property-storage definition
private String name;
```

- **Методи (methods)** – от главата "Методи", знаем, че методите са мястото в класа, където се описва поведението на обектите от този тип. В методите се изпълняват алгоритмите и се обработват данните на обекта.

Ето как изглежда един клас, който сме дефинирали сами и който притежава елементите, които описахме току-що:

Dog.java

```
// Class declaration
class Dog { // Openening brace of the class body

   // Property-field definition
   private String name;

   // Constructor definition
   public Dog() {
      this.name = "Sharo";
   }

   // Constructor definition
   public Dog(String name) {
      this.name = name;
   }

   // Property getter-method definition
   public String getName() {
      return this.name;
   }

   // Property setter-method definition
   public void setName(String name) {
      this.name = name;
   }

   // Method definition
   public void bark() {
      System.out.printf("Dog %s said: Wow-wow!%n", name);
   }
} // Closing brace of the class body
```

Сега няма да обясняваме изложения код, тъй като подробна информация ще бъде дадена по време на обяснението как се декларира всеки един от елементите на класа.

Използване на класове и обекти

В главата "Създаване и използване на обекти", видяхме подробно как се създават нови обекти от даден клас и как биват използвани. Сега на кратко ще си припомним как ставаше това.

Как да използваме дефиниран от нас клас?

За да можем да използваме някой клас, първо трябва да създадем обект от тип този клас. За целта използваме ключовата дума **new** в комбинация с някой от конструкторите на класа. Това ще създаде обект от дадения тип.

За да можем да манипулираме новосъздадения обект, ще трябва да го присвоим на променлива от типа на обекта. По този начин в тази променлива ще бъде запазена връзка (референция) към него.

Чрез променливата, използвайки точкова нотация, можем да извикваме методите, getter и setter методите на обекта, както и да достъпваме полетата (член-променливите) му.

Пример – кучешка среща

Нека вземем примера от предходната секция на тази глава, където е дефинирахме класа, който описва куче – **Dog** и добавим метод **main()** към него. В него ще онагледим казаното току-що:

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);  System.out.print("Write
first dog's name: ");
   String firstDogName = input.nextLine();

   // Assign dog name with a constructor
   Dog firstDog = new Dog(firstDogName);
   System.out.print("Write second dog's name: ");
   Dog secondDog = new Dog();

   // Assign dog name with a property
   secondDog.setName(input.nextLine());

   // Create a dog with a default name
   Dog thirdDog = new Dog();

   Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };
   for (Dog dog : dogs) {
      dog.bark();
   }
}
```

Съответно изходът от изпълнението ще бъде следният:

```
Write first dog's name: Bobcho
Write second dog's name: Walcho
Dog Bobcho said: Wow-wow!
Dog Walcho said: Wow-wow!
Dog Sharo said: Wow-wow!
```

В този метод, с помощта на класа **Scanner**, получаваме имената на обектите от тип куче, които потребителят трябва да въведе в конзолата.

Присвояваме първия въведен низ на променливата **firstDogName**. След това използваме тази променлива при създаването на първия обект от тип **Dog** – **firstDog**, като я подаваме като параметър на конструктора.

Създаваме втория обект от тип **Dog**, без да подаваме низ за името на кучето на конструктора му. След това въвеждаме името на второто куче, чрез класа **Scanner**, и получената стойност директно подаваме на setter метода – **setName()**. Извикването на метода **setName()** става чрез точкова нотация, приложена към променливата, която пази референция към втория създаден обект от тип **Dog – secondDog.setName()**.

Когато създаваме третия обект от тип **Dog**, не подаваме име на кучето на конструктора, нито след това модифицираме подразбиращата се стойност "**Sharo**".

След това създаваме масив от тип **Dog**, като го инициализираме с трите обекта, които току що създадохме.

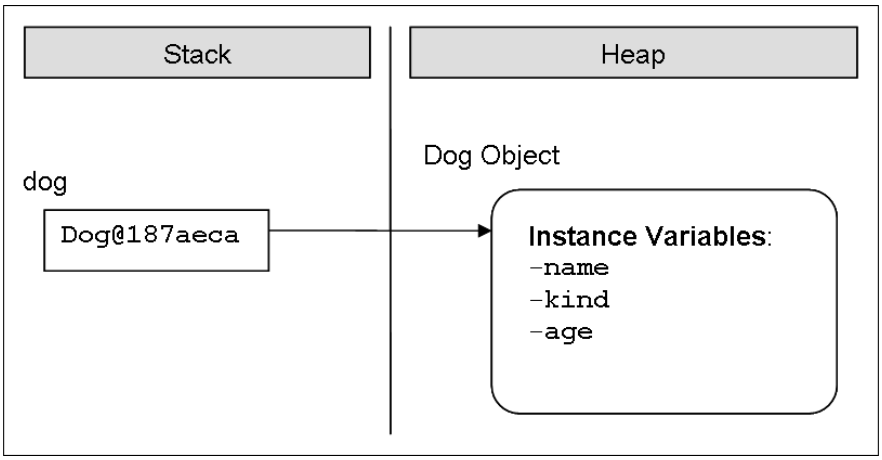
Накрая, използваме цикъл, за да обходим масива от обекти от тип **Dog**. На всеки елемент от масива, отново използвайки точкова нотация, извикваме метода **bark()** чрез **dog.bark()**.

Природа на обектите

Нека припомним също, че когато създадем един обект, той се състои от две части – реална част от обекта, която съдържа данните за конкретния обект и се намира в частта от оперативната памет, наречена хийп (heap) и референция към този обект (която се намира в друга част от оперативната памет, където се държат локалните променливи и параметрите на методите, наречена стек (stack)).

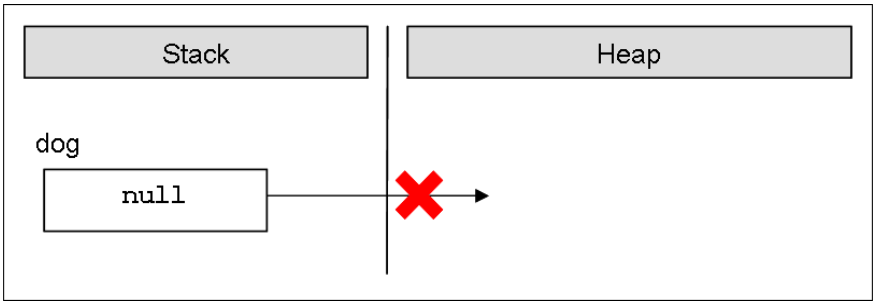
Например, нека имаме клас **Dog**, на който характеристиките му са име (name), порода (kind) и възраст (age). Създаваме променлива dog от този клас. Тази променлива се явява референция (указател) към обекта в динамичната памет (heap).

Референцията е променливата, чрез която достъпваме обекта. На схемата по-долу, примерната референция, която има връзка към реалния обект в хийпа, е с името **dog**. В нея, за разликата от променливите от примитивен тип, не се съдържа самата стойност (т.е. самият обект), а адрес към реалния обект в хийпа:



Ако се опитаме да отпечатаме стойността на една референция към обект в конзолата, ще получим нещо подобно на примера горе – **<class_name>@hex_digit**, което за нас е непонятно, но чрез него, Java държи връзка към реалния обект.

Когато декларираме една променлива от тип, някакъв клас, но не искаме тя да е инициализирана с връзка към конкретен обект, тогава трябва да й присвоим стойност **null**. Ключовата дума **null** в езика Java означава, че една променлива не сочи към нито един обект (липса на стойност):



Съхранение на собствени класове във файлове

Както знаем, всеки клас в Java се съхранява във файл с разширение **.java**. Файлът трябва да отговаря на определени изисквания, които ще изложим в следващите подсекции.

Вътрешна организация на .java файловете

Съгласно конвенцията на Sun за форматиране на кода, на първо място, във всеки един **.java** файл, трябва да декларираме пакета на класа. Това трябва да стане на първия ред във файла, който не е коментар. Както знаем от главата "Създаване и използване на обекти", не сме задължени да дефинираме класовете си в пакети, но е добра практика да го правим, тъй

като разпределянето на **.java** файловете помага за по-добрата организираност на кода и разграничаването на класовете с еднакви имена.

След декларирането на пакета, следва включването на пакети, които са нужни за работата на нашите класове, т.нар. **import**-дефиниции. По подобие на пакетите и те не са задължителни, но ако ги имаме в кода си, трябва да ги поставим незабавно след декларацията на пакета. Ако във файла няма деклариран пакет, тогава включването на пакети трябва да е на първия ред от файла.

След включването на пакети, следват декларациите на класовете, които могат да се съдържат в дадения **.java** файл. В един **.java** файл може да бъде деклариран повече от един клас. Декларацията на класовете става последователно, като редът за деклариране им няма значение. Въпреки всичко, заради прегледност, е по-добре всеки клас да се съхранява в отделен **.java** файл.

Ако не декларираме нито пакет в **.java** файла, нито включване на външни пакети, декларацията на първия клас от файла трябва да е на първия ред. Разбира се, ако не декларираме нито един клас в **.java** файла, това няма да бъде сметнато за грешка от компилатора, но това така или иначе няма смисъл.

Ето схема, по която трябва да се ориентираме, когато декларираме клас:

InternalJavaFileOrder.java

```
// Package definition – optional
package <package_name>;

// Import definitions - optional
import <package>;
import <package>;

// Class declaration
class <fist_class_name> {
   // ... Code ...
}

// Class declaration
class <second_class_name> {
   // ... Code ...
}

// ...

// Class declaration
class <n-th_class_name> {
   // ... Code ...
```

```
}
```

Декларирането на пакета и съответно включването на пакети са вече обяснени в главата "Създаване и използване на обекти". За това как се декларират класове, ще говорим след малко. Тук бяха представени формално, за да придобием представа за структурата на всеки **.java** файл.

Преди да продължим, да обърнем внимание на първия ред от схемата. Вместо декларация на пакет, има коментар. Това не е проблем, тъй като по време на компилация, коментарите се "изчистват" от кода и на първи ред от файла остава декларацията на пакета.

Кодиране на файловете. Четене на кирилица и Unicode

Когато създаваме **.java** файл, в който да дефинираме класа си, е добре да помислим за кодирането при съхраняването му на файловата система.

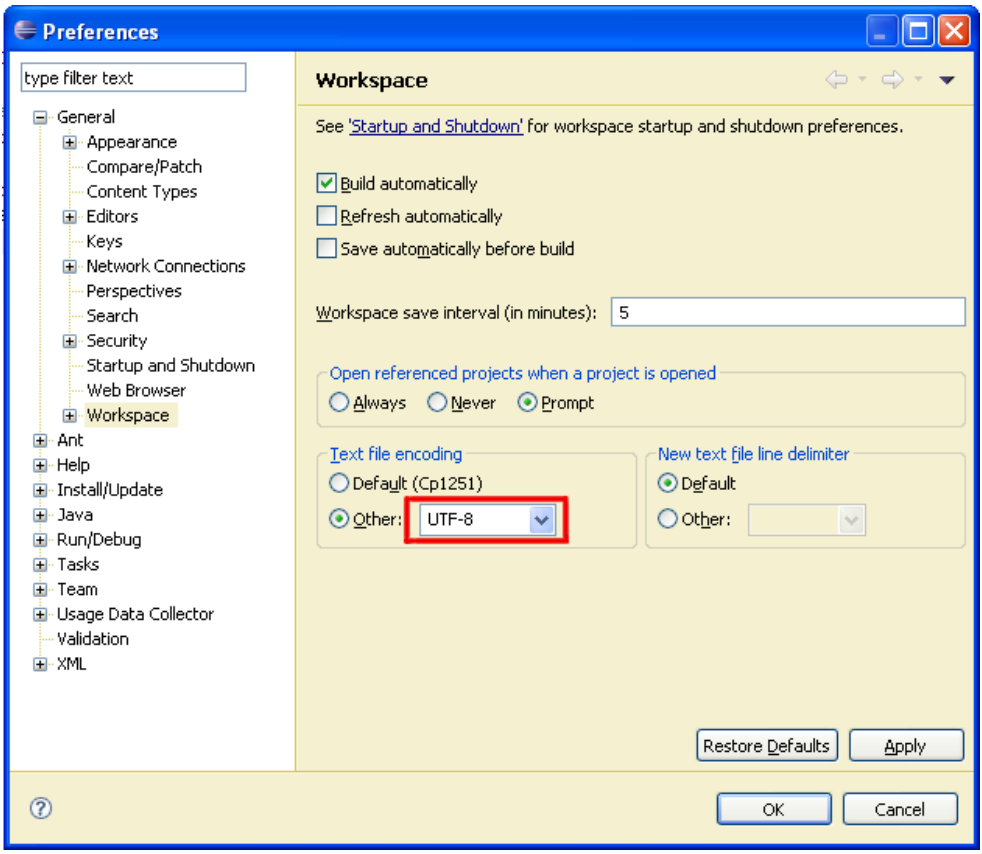
Вътрешно, Java представя кода в Unicode кодиране, затова, няма проблеми, ако във файла използваме символи, които са от азбуки, различни от латинската, например на кирилица:

EncodingTest.java

```
public class EncodingTest {

   // Тестов коментар
   static int години = 4;

   public static void main(String[] args) {
      System.out.println("години: " + години);
   }
}
```

Този код ще се компилира и изпълни без проблем, но за да запазим символите четими в редактора, трябва да изберем подходящото кодиране на файла.

По подразбиране, Eclipse използва настройките на операционната система за кодиране на **.java** файловете. Но ако при тази ситуация, във файла въведем по-особен символ, например китайски йероглиф, той няма да има представяне в кодовата таблица на кирилицата и най-вероятно няма да бъде визуализиран коректно в Eclipse. За да нямаме подобни проблеми, е добре да настроим Eclipse, да съхранява **.java** файлове с кодиране **UTF-8**. Това става по следния начин: избираме от менюто **Window -> Preferences -> General -> Workspace -> Text File Encoding -> Other -> UTF-8**:



Въпреки, че имаме възможността да използваме символи от други азбуки, в **.java** файловете, е препоръчително да пишем всички идентификатори и коментари на английски език, за да може кодът ни да е разбираем за повече хора по света.

Представете си само ако ви се наложи да дописвате код, писан от виетнамец, където имената на променливите и коментарите са на виетнамски език. Не искате да ви се случва, нали? А как ще се почувства един виетнамец, ако види променливи и коментари на български език?

Модификатори и нива на достъп (видимост)

Нека си припомним, от главата "Методи", че модификатор наричаме ключова дума, с помощта, на която даваме допълнителна информация на компилатора за кода, за който се отнася модификатора.

В Java има три модификатора за достъп. Те са **public**, **protected** и **private**. В тази глава ще се занимаем подробно само с **public** и **private**. Повече за **protected** ще научим в главата "Принципи на обектно-ориентираното програмиране".

Модификатори за достъп могат да се използват само пред следните елементи на класа: декларация, полета/свойства и методи на класа.

Модификатори и нива на достъп

Както казахме, в Java, има три модификатора за достъп – **public**, **protected** и **private**. С тях ние ограничаваме или позволяваме достъпа (видимостта) до елементите на класа, пред които те са поставени. На всеки един от тях, съответства ниво на достъп, което носи името на съответния модификатор, съответно – **public**, **protected** и **private**.

В Java обаче, има четвърто ниво на видимост до елемент на клас, за което няма модификатор за достъп. В литературата, това ниво на достъп се нарича **default** или **package** (след малко ще видим защо), а понякога в по-стари книги за Java се среща и като **friendly**. Съответно, класове, полета, свойства или методи на даден клас, които нямат модификатор за видимост в декларацията си, считаме, че имат ниво на достъп **default**.

В Java има три модификатора за видимост, но четири нива на достъп. Четвъртото, се нарича ниво на достъп default, и е в сила, когато пред съответния елемент на класа няма никакъв модификатор за достъп.

Сега ще дадем общо обяснение за различните нива, а в последствие, когато разглеждаме всеки един от елементите на класа, ще дадем по-подробна информация за всяко едно от тях.

Ниво на достъп public

Използвайки модификатора **public**, ние указваме на компилатора, че елементът, пред който е поставен, може да бъде достъпен от всеки друг клас, независимо дали е в текущия пакет или извън него. Той определя най-малко ограничителното ниво на видимост от всички нива в Java.

Ниво на достъп default

Това ниво на достъп се прилага, когато не се използва никакъв модификатор за достъп пред съответния елемент.

То е по-ограничително от **public**-видимостта, тъй като позволява, да достъпваме съответният елемент, само от класове, които се намират в същия пакет, в който се намира класът, на който принадлежи елементът.

Ниво на достъп private

Това е нивото на достъп, което налага най-голяма рестрикция на видимостта на класа и елементите му. Модификаторът **private** служи за

индикация, че елементът, за който се отнася, не може да бъде достъпван от никой друг клас, дори този клас да се намира в същия пакет.

Деклариране на класове

Декларирането на клас има строго определени правила (синтаксис), които трябва да спазваме:

```
[<access_modifier>] class <class_name>
```

Когато декларираме клас, задължително трябва да използваме ключовата дума **class**. След нея трябва да стои името на класа **<class_name>**.

Задължителните елементи от декларацията на класа са ключовата дума class и името на класа.

Освен ключовата дума **class** и името на класа, в декларацията на класа могат да бъдат използвани някои модификатори. Тук, ще обърнем внимание само на позволените модификатори за достъп.

Видимост на класа

Нека имаме два класа – **A** и **B**. Казваме, че класът **A**, има достъп до класа **B**, ако може да прави едно от следните неща:

- Създава обект (инстанция) от тип класа **B**.
- Достъпва определени методи и член-променливи (полета) в класа **B**, в зависимост от нивото на достъп на съответните методи и полета.

(Има и трета операция, която може да бъде извършвана с класове, когато имат видимост, наречена **наследяване на клас**, но за нея ще говорим подробно в главата "Принципи на обектно-ориентираното програмиране").

Както разбрахме, достъп означава "видимост". Ако класът **A** не може да "види" класа **B**, нивото на достъп на методите и полетата в класа **B** нямат значение.

Нивата на достъп, които един клас може да има са само **public** и **default**.

Ниво на достъп public

Ако декларираме един клас с модификатор за достъп **public**, ще можем да го достъпваме от всички Java класове, от всички пакети, независимо къде се намират те. Това означава, че всеки друг клас ще може да създава обекти от тип този клас и да има достъп до методите и полетата (с подходящо ниво на достъп) на класа.

Не трябва да забравяме, че ако искаме да използваме клас с ниво на достъп **public**, от друг пакет, различен от текущия, в който създаваме класа си,

трябва да използваме конструкцията за включване на пакети **import**, за пакета, в който се намира желаният за употреба клас.

Ако в един **.java** файл дефинираме повече от един клас, то в този файл е позволено да имаме само един единствен клас, който е с модификатор **public**. Освен това, когато декларираме един клас като **public**, файлът, в който се намира класът, трябва да има същото име, като на този клас.

Всеки клас с модификатор за достъп `public` трябва да е дефиниран в собствен файл с разширение `.java`, да е единствения public клас във файла и този файл трябва да е именуван с името на `public` класа.

Ниво на достъп default

В случай, че класът няма модификатор за достъп (т.е. има ниво на достъп **default**), този клас ще е видим само за класовете, които се намират в пакета, в който е деклариран класа. Затова понякога това ниво на достъп се нарича **package**.

Например, ако имаме клас **A** и клас **B**, но в различни пакети, и съответно клас **A** е с ниво на достъп **default**, то в клас **B** няма да можем да създаден нито един обект от тип **A**, или дори да дефинираме променлива от тип **A**.

A.java

```
package package1;

class A {}
```

Всъщност за класа **B**, класът **A** изобщо не съществува и ако въпреки всичко, се опитаме да използваме класът **A** в класа **B**, например:

B.java

```
package package2;

import package1.A;

public class B {

   public static void main(String[] args) {
      A aInstance;
   }
}
```

Компилаторът ще се "оплаче" със съобщение подобно на следното:

```
The type package1.A is not visible
```

Ниво на достъп private

За да сме изчерпателни, ще споменем, че като модификатор за достъп до клас, може да се използва модификатора за видимост **private**, но това е свързано с понятието "вътрешен клас" (inner class), което ще разгледаме в секцията "Вътрешни, локални и анонимни класове".

Тяло на класа

Преди да приключим с обяснението за декларация на клас ще кажем, че по подобие на методите, след декларацията на класа, следва неговото тяло, т.е. частта от класа, в която се съдържа програмния код:

```
[<access_modifier>] class <class_name> {

   // ... Class body – the code of the class goes here ...

}
```

Тялото на класа, започва с отваряща фигурна скоба "**{**" и завършва със затваряща – "**}**". Класът винаги трябва да има тяло.

Правила при създаването на име на клас

По подобие на деклариране на име на метод, за създаването на име на клас, има правила, които са препоръчани от Sun:

- Името на класа винаги започва с главна буква.
- Ако името на класа е съставено от няколко думи, първата буква от всяка нова дума, долепена до предходната, трябва да бъде главна.
- За имена на класове обикновено се използват съществителни имена.
- Името на класа е добре да бъде на английски език.

Ето няколко пример за имена на класове, които са правилно декларирани и форматирани:

```
Dog
Account
Car
BufferedReader
```

Ключовата дума this

Ключовата дума **this**, в Java, е референция към текущия обект – обектът, чийто метод или конструктор бива извикван. Можем я приемем като указател (референция), дадена ни априори от създателите на Java, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

```
this.myField
this.doMyMethod()
this(3,4) // if there is constructor with two int parameters
```

В момента няма да обясняваме изложения код. Разяснения ще дадем по-късно, в местата от секциите на тази глава, посветени на елементите на класа (полета, методи, конструктори) и засягащи ключовата дума **this**.

Полета

Както казахме в началото на главата, когато декларираме клас, ние описваме обект от реалния живот. За описанието на този обект, ние се фокусираме само върху характеристиките му, които имат отношение към проблема, който ще решава нашата програма.

Тези характеристики на реалния обект, ги интерпретираме в декларацията на класа, като декларираме набор от специален тип променливи, наречени полета, в които пазим данните за отделните характеристики. Когато създадем обекти по описанието на нашия клас, стойностите на полетата, ще съдържат конкретните характеристики, с които даден екземпляр (обект) се отличава от всички останали обекти от дадения клас.

Деклариране на полета в даден клас

До момента, сме се сблъсквали само с два типа променливи (вж. главата "Методи"), в зависимост от това къде са декларирани:

- **Локални променливи** – това са променливите, които са дефинирани в тялото на някой метод (или блок).
- **Параметри** – това са променливите в списъка с параметри, който един метод може да има в реда, на който е деклариран.

В Java съществува и трети вид променливи, наречени **полета** (**fields**) или **член-променливи на класа** (**instance variables**).

Те се декларират в тялото на класа, но извън тялото на блок, метод или конструктор (какво е конструктор, ще разгледаме подробно след малко).

Полетата се декларират в тялото на класа, но извън тялото на метод, конструктор или блок.

Ето един примерен код, в който се декларират различни полета:

MyClass.java

```
class MyClass {
   int age;
   long distance;
   String[] names;
   Dog myDog;
}
```

Формално, декларацията на полетата става по следния начин:

```
[<modifiers>] <field_type> <field_name>;
```

<field_type> определя типа на даденото поле. Той може да бъде, както примитивен тип (**byte**, **short**, **char** и т.н.) или масив, така и от тип, някакъв клас (например **String**).

<field_name> е името на даденото поле. Както при имената на обикновените променливи, когато именуваме една член-променлива трябва да спазваме правилата за идентификатори в Java (вж. главата "Примитивни типове и променливи").

<modifiers> е понятие, с което сме означили, както модификаторите за достъп, така и други модификатори. Те не са задължителна част от декларацията на едно поле.

Модификаторите и нивата за достъп, позволени в декларацията на едно поле са обяснени в секцията "Видимост на полета и методи" малко по-долу.

В тази глава, от другите модификатори, които не са за достъп, и могат да се използват при деклариране на полета на класа, ще обърнем внимание само на **static** и **final**. Оставащите модификатори (**transient** и **volatile**) са извън обсега на тази книга и няма да бъдат разглеждани.

Област на действие (scope)

Трябва да знаем, че **областта на действие** (**scope**) на едно поле е от реда, на който е декларирано, до затварящата фигурна скоба на тялото на класа.

Инициализация по време да деклариране

Когато декларираме едно поле е възможно едновременно с неговата декларация да му дадем първоначална стойност. Начинът, по който става това е същият, както при инициализацията (даването на стойност) на обикновена локална променлива:

```
[<modifiers>] <field_type> <field_name> = <initial_value>;
```

Разбира се, трябва **<initial_value>** да бъде от типа на полето, или някой съвместим с него тип. Например:

MyClass.java

```
class MyClass {

   int age = 5;
   long distance = 234; // the literal 234 is of integer type

   String[] names = new String[] { "Pencho", "Marincho" };
   Dog myDog = new Dog();

   // ... Other code ...
}
```

Стойности по подразбиране на полетата

Всеки път, когато създаваме нов обект от даден клас, виртуалната машина на Java, автоматично заделя за всяко поле от класа, памет в хийпа (heap – част от оперативната памет на компютъра, където се съхраняват обектите и полетата им). След като бъде заделена, тази памет се инициализира автоматично с подразбиращи стойности за конкретния тип поле. Това става, независимо дали след това, полето се инициализира изрично от програмиста на реда на неговата декларация или не.

Полетата се инициализират с подразбиращите стойности за типа им всеки път, когато нов обект от дадения тип бива създаван, независимо дали по време на декларацията им, изрично им се присвоява стойност или не.

Ето и списък с подразбиращите се стойности за всеки един тип:

Тип на поле	Стойност по подразбиране
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
boolean	false
char	"\u0000"

референция към обект	**null**

Например, ако създадем клас **Dog** и за него дефинираме полета име (**name**), възраст (**age**), дължина (**length**) и дали кучето е от мъжки пол (**isMale**), без да ги инициализираме по време на декларацията им:

Dog.java

```
public class Dog {

   String name;
   int age;
   int length;
   boolean isMale;

   public static void main(String[] args) {
      Dog dog = new Dog();

      System.out.println("Dog's name is: " + dog.name);
      System.out.println("Dog's age is: " + dog.age);
      System.out.println("Dog's length is: " + dog.length);
      System.out.println("Dog's is male: " + dog.isMale);
   }
}
```

Съответно, като резултат ще получим:

```
Dog's name is: null
Dog's age is: 0
Dog's length is: 0
Dog's is male: false
```

В примера по-горе, чрез точкова нотация (поставяне на точка между името на променливата и полето на обекта – вж. главата "Създаване и използване на обекти"), достъпваме полетата на обекта **dog**, от тип **Dog**, и съответно извеждаме резултата за всяко едно от тях. Както виждаме, след създаването на обекта, полетата се инициализират със стойностите им по подразбиране, които изброихме в таблицата с подразбиращите се стойности.

Разлика между локални променливи и полета

Ако дефинираме дадена локална променлива в един метод, без да я инициализираме, и веднага след това се опитаме да я използваме (да кажем отпечатаме стойността ѝ), това ще предизвика грешка при компилация, тъй като локалните променливи не се инициализират с подразбиращи се стойности по време на тяхното деклариране.

За разлика от полетата, локалните променливи, не биват инициализирани с подразбираща се стойност след тяхното деклариране.

Например:

```
public static void main(String[] args) {
   int notInitializedLocalVariable;
   System.out.println(notInitializedLocalVariable);
}
```

Кодът няма да може да се компилира и съобщението за грешка ще бъде подобно на следното:

```
The local variable notInitializedLocalVariable may not have been
initialized.
```

Собствени стойности по подразбиране

Добър стил на програмиране е обаче, когато декларираме полетата на класа си, изрично да ги инициализираме с дадена подразбираща се стойност. Макар и да сме сигурни, че Java ще даде подразбираща се стойност на всяко едно от полетата, ако инициализираме всяко едно от тях, всеки, който погледне нашия код, ще знае каква е първоначалната стойност на полето. Това подобрява четимостта на кода.

Пример за такова инициализиране може да дадем като модифицираме примера от предходната секция "Инициализация по време на деклариране", класът **MyClass**:

MyClass.java

```
class MyClass {

   int age = 0;
   long distance = 0;
   String[] names = null;
   Dog myDog = null;

   // ... Other code ...
}
```

Модификатор final

Както споменахме в началото на тази секция, в декларацията на едно поле е позволено да се използва модификатор **final**. Той не е модификатор за достъп, се използва за еднократно инициализиране на полета. След като

дадем стойност на едно поле, което има модификатор **final** в декларацията си, след това полето не може да приеме друга стойност.

Нека онагледим казаното с пример:

FinalModifierTest.java

```
class FinalModifierTest {
   final int age = 5; // Here is the final field

   public static void main(String[] args) {
      FinalModifierTest instance = new FinalModifierTest();

      instance.age = 15; // Trying to reinitialize it...
   }
}
```

Ако се опитаме да компилираме този клас, няма да успеем, като ще получим следното съобщение за грешка:

```
The final field FinalModifierTest.age cannot be assigned.
```

Решението е или да махнем **final** от декларацията на полето или да не се опитваме да го инициализираме отново.

Методи

В главата "Методи" подробно се запознахме с това как да декларираме и използваме метод. В тази секция, накратко ще припомним казаното там и ще се фокусираме върху някои нови особености при декларирането и създаването на методи.

Деклариране на методи в даден клас

Декларирането на методи, както знаем става по следния начин:

```
[<modifiers>] <return_type> <method_name>([<parameters_list>]) {

   // ... Method"s body ...

   [<return_statement>];
}
```

Задължителните елементи при деклариране на метода са типът на връщаната стойност **<return_type>**, името на метода **<method_name>** и отварящата и затварящата кръгли скоби – "**(**" и "**)**".

Списъкът от параметри **<params_list>** не е задължителен. Използваме го да подаваме информация на метода, който декларираме, ако той се нуждае от такава.

Знаем, че ако типът на връщаната стойност **<return_type>** е **void**, тогава **<return_statement>** може да участва само с оператора **return**, с цел прекратяване действието на метода. Ако **<return_type>** е различен от **void**, методът задължително трябва да връща резултат чрез ключовата дума **return**, като резултатът е от тип **<return_type>** или съвместим с него.

Реалната работа, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "**{**" и "**}**".

Макар, че разгледахме някои от модификаторите за достъп, позволени да се използват при декларирането на един метод, в секцията "Видимост на полета и методи" ще разгледаме по-подробно тази тема.

Ще разгледаме модификатора **static** в последната секция на тази глава.

Пример – деклариране на метод

Нека погледнем декларирането на един метод за намиране сбор на две цели числа:

```
int add(int number1, int number2) {
   int result = number1 + number2;
   return result;
}
```

Името, с което сме го декларирали, е **add()**, а типът на връщаната му стойност е **int**. Списъкът му от параметри се състои от два елемента – променливите **number1** и **number2**. Съответно, връщаме стойността на сбора от двете числа като резултат.

Достъп до нестатичните данни на класа

В главата "Създаване и използване на обекти", видяхме как чрез оператора точка, можем да достъпим полетата и да извикаме методите на един клас. В тази секция ще разгледаме как можем по подобен начин да достъпваме полета и да извикваме методи на даден клас, които не са статични, т.е. нямат модификатор **static**, в декларацията си.

Например, нека имаме клас **Dog**, с поле за възраст – **age**. За да отпечатаме стойността на това поле, е нужно да създадем обект от клас **Dog** и да достъпим полето на този обект, чрез точкова нотация**:**

Dog.java

```
public class Dog {
```

```
   int age = 2;

   public static void main(String[] args) {
      Dog dog = new Dog();
      System.out.println("Dog's age is: " + dog.age);
   }
}
```

Съответно резултатът ще бъде:

```
Dog's age is: 2
```

В следващите подсекции ще разберем как това става в рамките на самия клас.

Достъп до нестатичните полетата на класа от нестатичен метод

По-късно, в тази глава ще научим, че достъпа до стойността на едно поле, да се осъществява, не директно – чрез оператора точка (както бе в последния пример **dog.age**), а чрез метод. Нека използваме това знание предварително и в класа **Dog**, си създадем нестатичен метод, който при извикване, връща стойността на полето **age**:

```
int getAge() {
   return this.age;
}
```

Както виждаме, за да достъпим стойността на полето за възрастта, вътре, от самия клас, използваме ключовата дума **this**. Знаем, че ключовата дума **this** е референция към текущия обект, към който се извиква метода. Следователно, в нашия пример, с "**return this.age**", ние казваме "от текущия обект (**this**), вземи (използването на оператора точка), стойността на полето **age** и го върни като резултат от метода (чрез ключовата дума **return**) ". Тогава, вместо в метода **main()** да достъпваме стойността на полето **age** на обекта **dog**, ние просто ще извикаме метода **getAge()**:

```
public static void main(String[] args) {
   Dog dog = new Dog();
   System.out.println("Dog's age is: " + dog.getAge());
}
```

Резултатът след тази промяна, ще бъде отново същият.

Формално, декларацията за достъп до поле в рамките на класа, е следната:

```
this.<field_name>
```

Нека подчертаем, че този достъп е възможен, само от нестатичен код, т.е. метод или блок, който няма модификатор **static**.

Освен за извличане на стойността на едно поле, можем да използваме ключовата дума **this**, също така за модифициране.

Например, нека декларираме метод **getOlder()**, който извикваме всяка година на датата, на рождения ден на нашия домашен любимец и който, увеличава възрастта му с една година:

```
void getOlder() {
   this.age++;
}
```

За да проверим дали това, което написахме работи коректно, в края на метода **main()** добавяме следните два реда:

```
// One year later, on the birthday date...
dog.getOlder();
System.out.println("After one year dog's age is: " + dog.age);
```

След изпълнението, резултатът е следният:

```
Dog's age is: 2
After one year dog's age is: 3
```

Извикване нестатичните методи на класа от нестатичен метод

По подобие на полетата, които нямат **static** в декларацията си, методите, които също не са статични, могат да бъдат извиквани в тялото на класа, чрез ключовата дума **this**. Това става, след като към нея, чрез точкова нотация извикаме метода, който ни е необходим:

```
this.<method_name>(...)
```

Например, нека създадем метод **printAge()**, който отпечатва възрастта на обекта от тип **Dog**, като за целта извиква метода **getAge()**:

```
void printAge() {
   int myAge = this.getAge(); // Calling getAge() by this
   System.out.println("My age is: " + myAge);
}
```

На първия ред от примера, указваме, че искаме да получим възрастта (стойността на полето **age**) на текущия обект, извиквайки метода **getAge()**, на текущия обект. Това става, чрез ключовата дума **this**.

Достъпването на нестатичните елементи на класа (полета и методи), се осъществява чрез ключовата дума `this` и оператора за достъп – точка.

Достъп до нестатични данни на класа без използване на this

Когато достъпваме полетата на класа или извикваме нестатичните му методи, е възможно, да го направим без ключовата дума **this**. Тогава двата метода, които декларирахме могат да бъдат записани по следния начин:

```
int getAge() {
   return this.age;
}

void getOlder() {
   this.age++;
}
```

Ключовата дума **this** се използва, **изрично** да укаже, че правим достъп до нестатично поле на даден клас или извикваме негов нестатичен метод. Когато това не е необходимо може да бъде пропускана и директно да се достъпва елементът на класа.

Когато не е нужно изрично да се укаже, че правим достъп до елемент на класа, ключовата дума `this`, може да бъде пропусната.

Припокриване на полета и локални променливи (scope overlapping)

От секцията "Деклариране на полета в даден клас" по-горе, знаем, че областта на действие на едно поле е от реда, на който е декларирано полето, до затварящата скоба на тялото на класа. Например:

OverlappingScopeTest.java

```
class OverlappingScopeTest {
   int myValue = 3;

   void printMyValue() {
```

```
      System.out.println("My value is: " + myValue);
   }

   public static void main(String[] args) {
      OverlappingScopeTest instance = new OverlappingScopeTest();

      instance.printMyValue();
   }
}
```

Този код ще изведе в конзолата като резултат:

```
My value is: 3
```

От друга страна, когато имплементираме тялото на един метод, ни се налага да дефинираме локални променливи, които да използваме по време на изпълнение на метода. Както знаем, областта на действие на тези локални променливи започва от реда, на който са декларирани и продължава до затварящата фигурна скоба на тялото на метода. Например, нека добавим този метод в току що декларирания клас **OverlappingScopeTest**:

```
int calculateNewValue(int newValue) {
   int result = myValue + newValue;
   return result;
}
```

Тук, локалната променлива, която използваме, за да изчислим новата стойност, е **result**.

Понякога обаче, може да се случи така, че името на някоя локална променлива да съвпадне с името на някое поле. Тогава настъпва колизия.

Нека първо погледнем един пример, преди да обясним за какво става въпрос. Нека модифицираме метода **printMyValue()** по следния начин:

```
void printMyValue() {
   // Defining new local variable with the same name
   int myValue = 5;

   System.out.println("My value is: " + myValue);
}
```

Ако декларираме така метода, дали той ще се компилира? А ако се компилира, дали ще се изпълни? Ако се изпълни коя стойност ще бъде отпечатана – тази на полето или тази на локалната променлива?

Така деклариран, след като бъде изпълнен метода **main()**, резултатът, който ще бъде отпечатан, ще бъде:

```
My value is: 5
```

Това е така, тъй като Java позволява да се дефинират локални променливи, чиито имена съвпадат с някое поле. Ако това се случи, казваме, че областта на действие на локалната променлива препокрива областта на действие на полето (scope overlapping).

Точно затова, областта на действие на локалната променлива **myValue** със стойност **5**, препокри областта на действие на полето със същото име. Тогава, при отпечатването на стойността, бе използвана стойността на локалната променлива.

Въпреки това, понякога се налага да бъде използвано полето, въпреки че употребата му е в областта на действие на някоя променлива със същото име. В този случай, за да извлечем стойността на полето, използваме ключовата дума **this**. За целта, достъпваме полето чрез оператора точка, приложен към **this**. По този начин еднозначно указваме на виртуалната машина, че искаме да използваме стойността на полето, не на локалната променлива, която има същото име.

Нека разгледаме отново нашия пример с извеждането на стойността на полето **myValue**:

```
void printMyValue() {
   int myValue = 5;

   // Accessing the field value by the keyword this
   System.out.println("My value is: " + this.myValue);
}
```

Този път, резултатът от извикването на метода е:

```
My value is: 3
```

Видимост на полета и методи

В началото на главата разгледахме общите положения с модификаторите и нивата на достъп на елементите на един клас в Java. По-късно се запознахме подробно с нивата на достъп при деклариране на един клас.

Сега ще разгледаме нивата на видимост на полетата и методите в класа. Тъй като полетата и методите са елементи на класа и имат едни и същи правила при определяне на нивото им на достъп, ще изложим тези правила едновременно.

За разлика от декларацията на клас, при деклариране на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – **public**, **protected**, **default** и **private**. Нивото на видимост **protected** няма да бъде

разглеждано в тази глава, тъй като е обвързано с тематиката на главата "Принципи на обектно-ориентираното програмиране" и ще бъде обяснено подробно в нея.

Преди да продължим, нека припомним, че ако един клас **A**, не е видим (няма достъп) от друг клас **B**, тогава нито един елемент (поле или метод) на класа **A**, не може да бъде достъпен от класа **B**.

Ако два класа не са видими един за друг, то елементите им (полета и методи) не са видими също, независимо с какви нива на достъп са декларирани самите те.

В следващите подсекции, към обясненията, ще разглеждаме примери, в които имаме два класа (**Dog** и **Kid**), които са видими един за друг, т.е. всеки един от класовете може да създава обекти от тип – другия клас и да достъпва елементите му, в зависимост от нивото на достъп, с което са декларирани. Ето какъв е кодът на класовете:

Dog.java

```
public class Dog {
   public String name = "Sharo";

   public String getName() {
      return this.name;
   }

   public void bark() {
      System.out.println("wow-wow");
   }

   public void doSth() {
      this.bark();
   }
}
```

и съответно:

Kid.java

```
public class Kid {

   public void callTheDog(Dog dog) {
      System.out.println("Come, " + dog.name);
   }

   public void wagTheDog(Dog dog) {
      dog.bark();
```

```
    }
}
```

В момента, всички елементи (полета и методи) на двата класа са декларирани с модификатор за достъп **public**, но при обяснението на различните нива на достъп, ще го променяме в зависимост от съответното ниво. Това, което ще ни интересува е как промяната в нивото на достъп на елементите (полета и методи) на класа **Dog** и ще рефлектира върху достъпа до тези елементи, когато този достъп се извършва от:

- Самото тяло на класа **Dog**.
- Тялото на класа **Kid**, съответно вземайки в предвид дали **Kid** е в пакета, в който се намира класа **Dog** или не.

Ниво на достъп public

Когато метод или променлива на класа са декларирани с модификатор за достъп **public**, те могат да бъдат достъпвани от други класове, независимо дали другите класове са декларирани в същия пакет или извън него.

Dog.java

D
```
public String name = "Sharo";

public String getName() {
    return this.name;
}
```
D
```
public void bark() {
    System.out.println("wow-wow");
}
public void doSth() {
    this.bark();
}
```

Kid.java

R
```
public void callTheDog(Dog dog) {
    System.out.println("Come, "
        + dog.name);
}
```
R
```
public void wagTheDog(Dog dog) {
    dog.bark();
}
```

Тук сме означили достъпа до елементите на класа **Dog**, съответно с:

D — Достъп до елемент на класа осъществен в самата декларация на класа

Достъп до елемент на класа осъществен, чрез референция към обект, създаден в тялото на друг клас

Както виждаме, без проблем осъществяваме, достъп до полето **name** и метода **bark()**, в класа **Dog**, от тялото на самия клас. Също така, независимо дали класът **Kid** е в пакета на класа **Dog**, можем от тялото му, да достъпим полето **name** и съответно да извикаме метода **bark()** чрез оператора точка, приложен към референцията **dog** към обект от тип **Dog**.

Ниво на достъп default

Когато елемент на някой клас бъде деклариран с ниво на достъп **default**, т.е. без модификатор за достъп, тогава този елемент на класа може да бъде достъпван от всеки клас в същия пакет, но не и за класовете извън пакета:

Dog.java


```
String name = "Sharo";

public String getName() {
    return this.name;
}
void bark() {
    System.out.println("wow-wow");
}
public void doSth() {
    this.bark();
}
```

Съответно, за класа **Kid**, разглеждаме двата случая:

- Когато е в **същия пакет**, достъпът до елементите на класа **Dog**, ще бъде позволен:

Kid.java


```
public void callTheDog(Dog dog) {
    System.out.println("Come, "
        + dog.name);
}

public void wagTheDog(Dog dog) {
    dog.bark();
}
```

- Когато класът **Kid** е **външен за пакета**, в който е деклариран класа **Dog**, тогава достъпът до полето **name** и метода **bark()** ще е невъзможен:

Kid.java

```
public void callTheDog(Dog dog) {
    System.out.println("Come, "
        + dog.name);
}

public void wagTheDog(Dog dog) {
    dog.bark();
}
```

Въпреки всичко, ако се опитаме да компилираме класа **Kid**, във втория случай, когато е външен за пакета, в който се намира класа **Dog**, няма да успеем и грешките, които ще бъдат изведени, ще бъдат следните:

```
The field Dog.name is not visible.
The method bark() from the type Dog is not visible.
```

Ниво на достъп private

Нивото на достъп, което налага най-много ограничения е **private**. Елементите на класа, които са декларирани с модификатор за достъп **private**, не могат да бъдат достъпвани от никой друг клас, различен от класа, в който са декларирани.

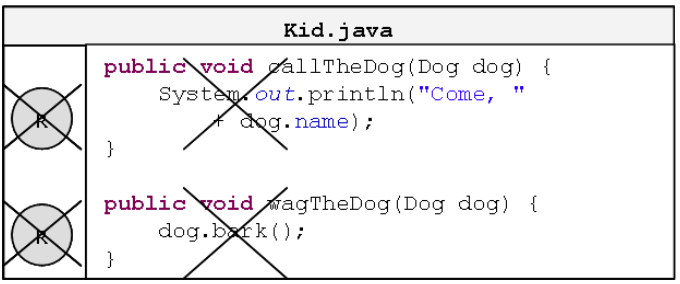
Това ще рече, че ако декларираме полето **name** и метода **bark()** на класа **Dog**, с модификатори **private**:

Dog.java

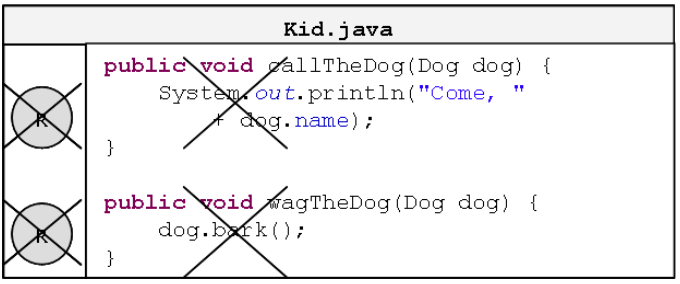
```
private String name = "Sharo";

public String getName() {
    return this.name;
}
private void bark() {
    System.out.println("wow-wow");
}
public void doSth() {
    this.bark();
}
```

Тогава, достъпът до тях, от тялото на класа **Kid**, няма да бъде достъпен, независимо, дали класът **Kid** е деклариран в пакета, в който е деклариран класа **Dog** или е вън от него:

- Когато **Kid** е в **същия пакет**, достъпът до полето **name** и метода **bark()** на класа **Dog**, няма да бъде позволен:



- Когато класът **Kid** е **външен за пакета**, в който е деклариран класът **Dog**, тогава достъпът до полето **name** и метода **bark()** отново ще е невъзможен:



Трябва да знаем, че когато полето ни има модификатор за достъп, най-често той е добре да бъде модификатор за достъп **private**, тъй като той дава възможно най-висока защита за достъп до стойността на полето. Съответно, достъпът и модификацията на тази стойност се осъществяват единствено чрез методи. Повече за тази техника ще научим в секцията "Капсулация (Encapsulation)" на главата "Принципи на обектно-ориентираното програмиране".

Как се определя нивото на достъп на елементите на класа?

Преди да приключим със секцията за видимостта на елементите на един клас, нека направим един експеримент. Нека в класа **Dog** полето **name** и метода **bark()** са декларирани с модификатор за достъп **private**. Нека също така, декларираме метод **main()**, със следното съдържание:

Dog.java

```
public class Dog {
   private String name = "Sharo";

   // ...

   private void bark() {
      System.out.println("wow-wow");
```

```
    }

    // ...

    public static void main(String[] args) {
        Dog myDog = new Dog();
        System.out.println("My dog's name is " + myDog.name);
        myDog.bark();
    }
}
```

Въпросът, който стои пред нас е, ще се компилира ли класът **Dog**, при положение, че сме декларирали елементите на класа с модификатор за достъп **private**, а в същото време ги извикваме с точкова нотация, приложена към променливата **myDog**, в метода **main()**?

Стартираме компилацията и тя минава **успешно**. Това е така, тъй като модификаторите за достъп до елементите на класа се прилагат на ниво клас, а не на ниво обекти, т.е. тъй като променливата **myDog** е дефинирана в тялото на класа **Dog**, можем да достъпваме елементите й (полета и методи) чрез точкова нотация, независимо че са декларирани с ниво на достъп **private**. Ако обаче се опитаме да направим същото от тялото на класа **Kid**, това няма да е възможно, тъй като достъпът до **private** полетата на класа няма да е разрешен.

Нивото на достъп на елемент от класа, се определя на ниво клас, а не на ниво обект от даден клас.

Съответно, резултатът от изпълнението на метода **main()**, който декларирахме в класа **Dog** ще бъде следния:

```
My dog's name is Sharo
wow-wow
```

Конструктори

В обектно-ориентираното програмиране, когато създаваме обект от даден клас, е необходимо да извикаме елемент от класа, наречен конструктор.

Какво е конструктор?

Конструктор на даден клас, наричаме псевдометод, който няма тип на връщана стойност, носи името на класа и който се извиква чрез ключовата дума **new**.

Задачата на конструктора е да задели памет в хийпа, където ще съхраняват данните, които се пазят в полетата на конкретния обект (тези, които не са

static), инициализира всяко поле с подразбиращата се за типа му стойност и връща референция към новосъздадения обект.

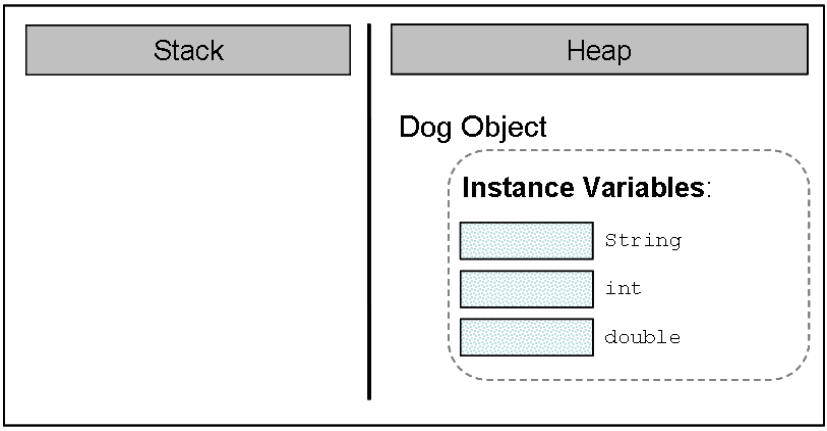
Извикване на конструктор

За разлика от методите, в Java, единствения начин да извикаме един конструктор е чрез използването на ключовата дума **new**.

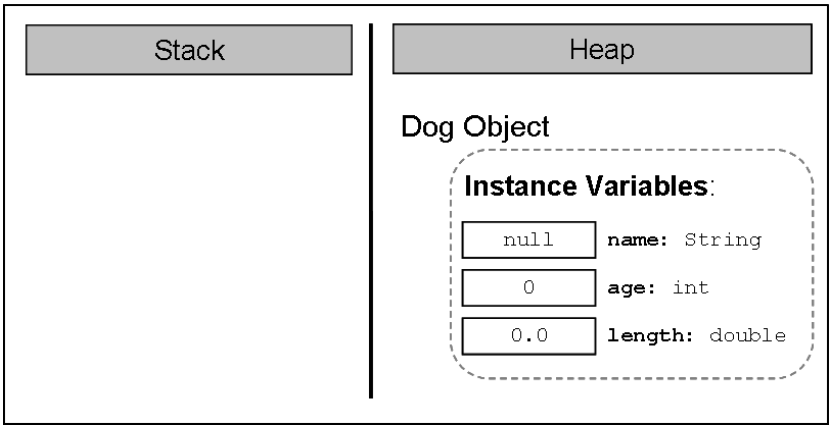
Нека разгледаме един пример, от който ще стане ясно как работи конструктора. От главата "Създаване и използване на обекти", знаем как се създава обект:

```
Dog myDog = new Dog();
```

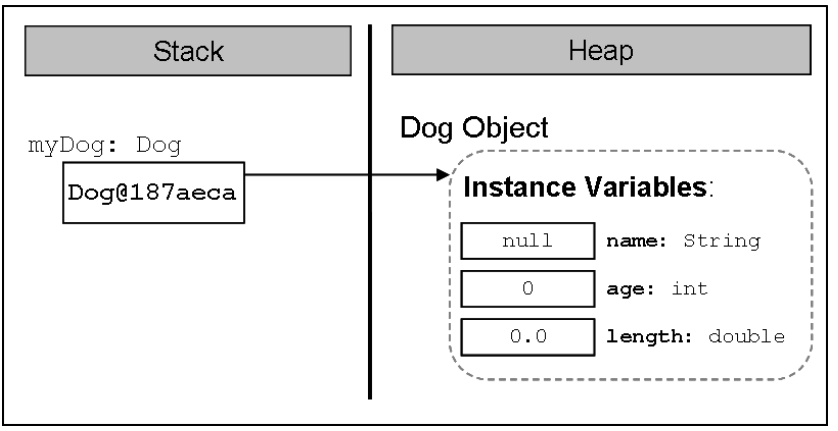
В случая, чрез ключовата дума **new**, стартираме конструктора на класа **Dog**. След това, той заделя паметта необходима за новосъздадения обект от тип **Dog:**



Инициализира полетата му, ако има такива, с подразбиращите се стойности, за съответните им типове:



Ако създаването на новия обект е завършило успешно, конструкторът връща референция към него, която ние присвояваме на променливата **myDog**, от тип – класа **Dog**:



Деклариране на конструктор

Ако имаме класа **Dog**, ето как би изглеждал неговия най-опростен конструктор:

```
public Dog() {  }
```

Формално, декларацията на конструктора изглежда по следния начин:

```
[<modifiers>] <class_name>([<parameters_list>])
```

Както вече казахме, конструкторите приличат на методи, но нямат тип на връщана стойност (затова ги нарекохме псевдометоди).

Име на конструктора

В Java, задължително, името на всеки конструктор съвпада с името на класа, в който го декларираме - **<class_name>**. В примера по-горе, името на конструктора е същото, каквото е името на класа – **Dog**. Трябва да знаем, че както при методите, името на конструктора винаги е следвано от кръгли скоби – "(" и ")".

Трябва да отбележим, че в Java е напълно легално, да се декларира метод, който притежава име, което съвпада с името на класа. Разбира се това не го прави конструктор, тъй като конструкторите нямат тип на връщаната стойност. Ето един такъв пример:

MyClass.java

```
public class MyClass {
```

```
   // LEGAL constructor
   public MyClass() {
   }

   // Misleading method - has return type
   String MyClass() {
      return "MyClass() method has finished successfully.";
   }

   public static void main(String[] args) {
      MyClass instance = new MyClass();

      // Calling the tricky method...
      System.out.println(instance.MyClass());
   }
}
```

Списък с параметри

По подобие на методите, ако за създаването на обекта, са необходими допълнителни данни, конструкторът ги получава чрез списък от параметри – **<parameters_list>**. В примерния конструктор на класа **Dog**, няма нужда от допълнителни данни за създаване на обект от такъв тип и затова няма деклариран списък от параметри. Повече за списъка от параметри ще разгледаме в една от следващите секции – "Деклариране на конструктор с параметри".

Разбира се след декларацията на конструктора, следва неговото тяло, което е като тялото на всеки един метод в Java.

Модификатори

Забелязваме, че в декларацията на конструктора, може да се добавят модификатори – **<modifiers>**. За модификаторите, които познаваме и които не са модификатори за достъп, т.е. **final** и **static**, трябва да кажем, че не са позволени за употреба при декларирането на конструктори.

Видимост на конструкторите

По подобие на полетата и методите на класа, конструкторите, могат да бъдат декларирани с нива на достъп **public**, **protected**, **default** и **private**. Нивото на достъп **protected**, ще бъде обяснено в главата "Принципи на обектно-ориентираното програмиране". За останалите нива на достъп, трябва да кажем, че видимостта на конструкторите е същата като тази на полетата и методите.

Ниво на достъп public

Когато конструкторът е с модификатор за достъп **public**, той може да бъде извикан от кой да е друг клас (стига видимостта на неговия собствен клас да го позволява). В контекста на примерите с класовете **Dog** и **Kid**, които използвахме в предните секции, ако нивото на достъп на целия клас **Dog** е **public**, и съответно конструкторът му също е с видимост **public**, обекти от тип **Dog**, могат да бъдат създавани в класа **Kid**, независимо **Kid** в кой пакет е деклариран:

Dog.java

```
public class Dog {
   public String name = "Sharo";

   public Dog() {  }

   // ... Rest of the class body ...
}
```

и съответно:

Kid.java

```
public class Kid {
   public static void main(String[] args) {

      // ... Constructor invocation ...
      Dog myDog = new Dog();

      System.out.println("My dog is called " + myDog.getName());
   }
}
```

Ниво на достъп default

Ако конструкторът е деклариран без модификатор за достъп, т.е. има ниво на достъп **default**, обекти от нашия клас могат да бъдат създавани само в рамките на класа ни, или в класовете, които се намират в пакета, в който се намира нашия клас.

С други думи, ако класът **Kid**, е в пакет **package2**, а класът **Dog**, се намира в пакет **package1** и съответно конструкторът на класа **Dog** е без модификатор за достъп:

Dog.java

```
package package1;
```

```
public class Dog {
   // ...

   Dog() { } // Constructor with Default Access Control

   // ...
}
```

Създаването на обекта, реферирan от променливата **myDog** в метода **main()** на класа **Kid**, ще е невъзможно:

Kid.java

```
public class Kid {
   public static void main(String[] args) {

      // ... Constructor invocation ...
      Dog myDog = new Dog(); // IMPOSSIBLE!

      // ...
   }
}
```

Съобщението за грешка, което компилаторът ще изведе, ще бъде следното:

```
The constructor Dog() is not visible.
```

Ниво на достъп private

Както можем да се досетим, когато модификаторът за достъп в декларацията на един конструктор е **private**, то можем да създаваме обекти от тип този клас, само в рамките на класа. Всички останали класове, независимо в кой пакет се намират, не могат да извикват конструктор с ниво на видимост **private**.

Ако в декларацията на конструктора на класа **Dog** е включен модификатора **private**:

```
private Dog() {}
```

Независимо в кой пакет е класа **Kid**, при опит да създадем обект от тип **Dog**, съобщението за грешка ще бъде същото, което бе указано в секцията за ниво на достъп **default**:

```
The constructor Dog() is not visible.
```

Инициализация на полета в конструктора

Както научихме по-рано, при извикването на конструктора, се заделя памет за полетата на обекта от дадения клас, които не са декларирани с модификатор за достъп **static**, като паметта за тези полета се инициализира със стойностите по подразбиране за съответния тип (вж. секция "Извикване на конструктор").

Освен това, чрез конструкторите най-често инициализираме полетата на класа, със стойности зададени от нас, а не с подразбиращите се за типа.

Например, в примерите, които разглеждахме до момента, винаги полето **name** на обекта от тип **Dog**, го инициализирахме по време на неговата декларация:

```
String name = "Sharo";
```

Вместо да правим това по време на декларацията на полето, по-добър стил на програмиране е да му дадем стойност в конструктора:

Dog.java

```
public class Dog {
   String name;

   public Dog() {
      this.name = "Sharo";
   }

   // ... The rest of the class body ...
}
```

В някои книги се препоръчва, въпреки че инициализираме полетата в конструктора, изрично да присвояваме подразбиращите се за типа им стойности по време на инициализация, с цел да се подобри четимостта на кода, но това е въпрос на личен избор:

Dog.java

```
public class Dog {
   private String name = null;

   public Dog() {
      this.name = "Sharo";
   }

   // ... The rest of the class body ...
}
```

Инициализация на полета в конструктора – представяне в паметта

Нека разгледаме подробно, какво прави конструкторът след като бъде извикан и в тялото му инициализираме полетата на класа. Знаем, че при извикване, той ще задели памет за всяко поле и тази памет бъде инициализирана със стойността по подразбиране.

Ако полетата са от примитивен тип, тогава след подразбиращите се стойности, ще бъдат присвоени новите, които ние подаваме.

В случая, когато полетата са от референтен тип, например нашето полето **name**, конструкторът ще ги инициализира с **null**. След това ще създаде обекта от съответния тип, в случая низа **"Sharo"** и накрая ще се присвои референция към новия обект в съответното поле, при нас – полето **name**.

Същото ще се получи, ако имаме и други полета, които не са примитивни типове и ги инициализираме в конструктора. Например, нека имаме клас, който описва каишка – **Collar**:

Collar.java

```
public class Collar {

   private int size;

   public Collar() {}
}
```

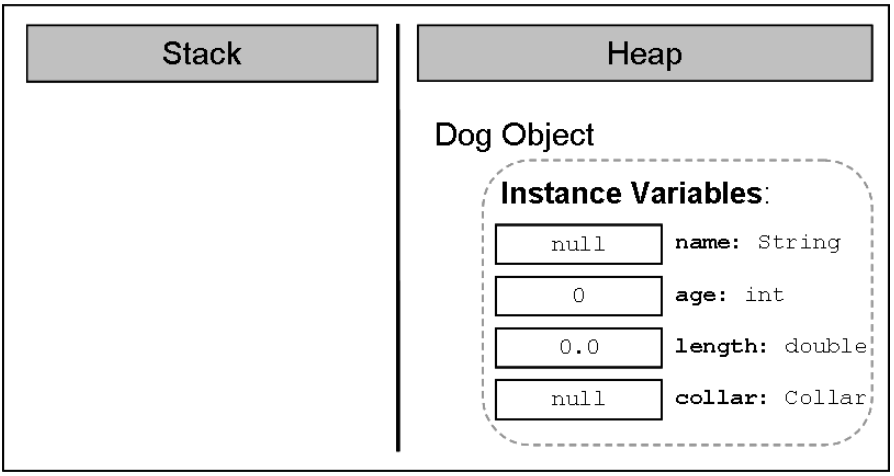
Нека съответно нашият клас **Dog**, има поле **collar**, което е от тип **Collar** и което инициализираме в конструктора на класа:

Dog.java

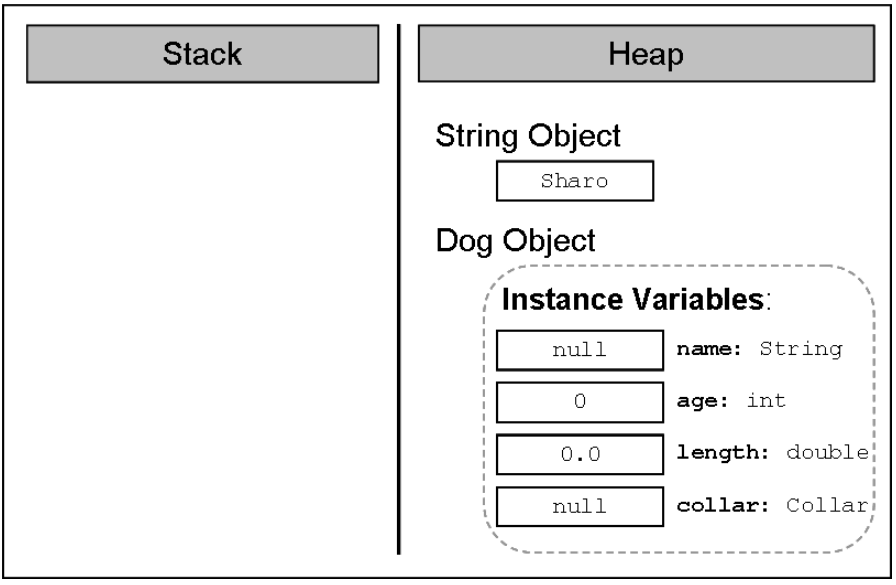
```
public class Dog {

   private String name;
   private int age;
   private double length;
   private Collar collar;

   public Dog() {
      this.name = "Sharo";
      this.age = 3;
      this.length = 0.5;
      this.collar = new Collar();
   }

   public static void main(String[] args) {
```

```
        Dog myDog = new Dog();
    }
}
```

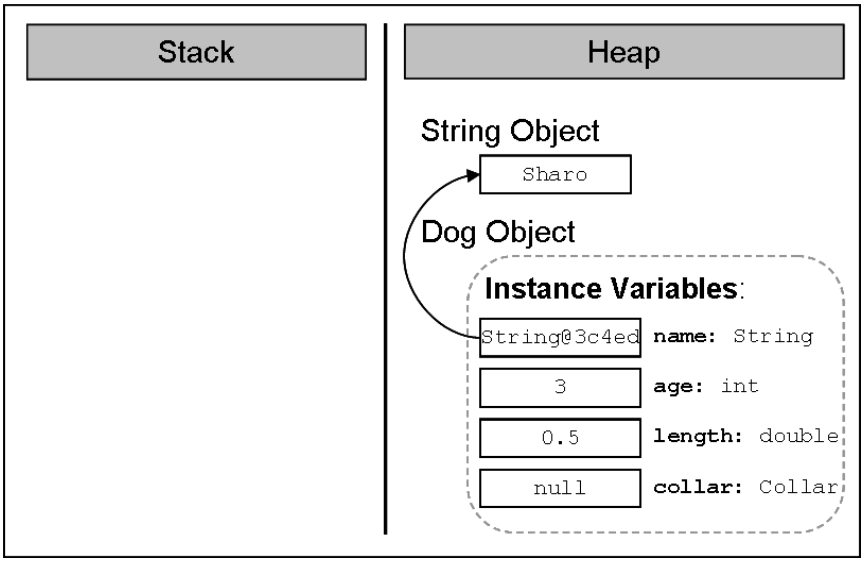
Нека проследим стъпките, през които минава конструктора, след като бъде извикан в **main()** метода. Както знаем, той ще задели памет в хийпа за всички полета, и ще ги инициализира със съответните им подразбиращи се стойности:



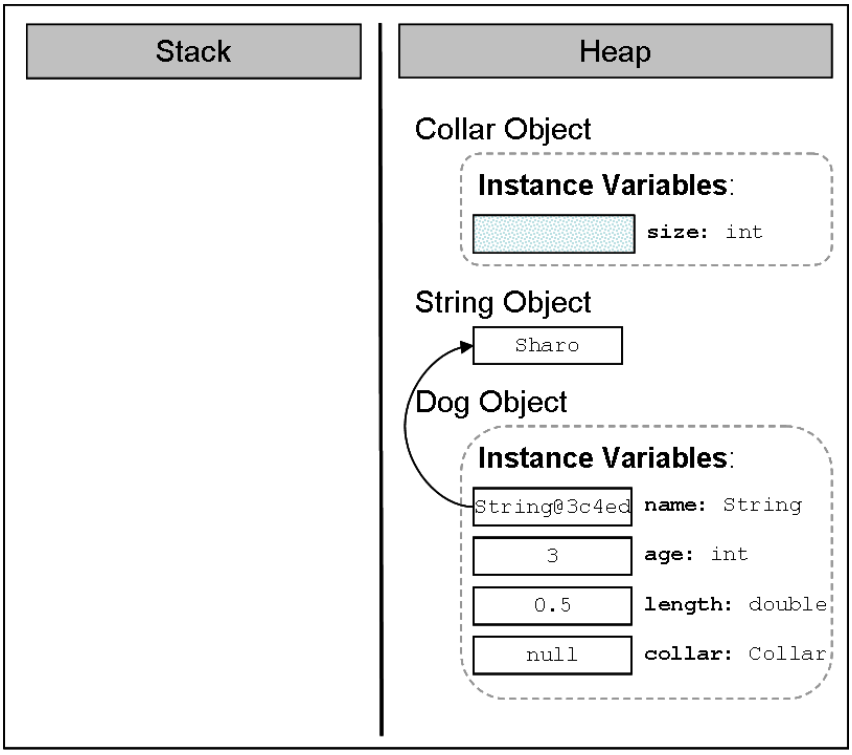
След това, конструкторът ще трябва да се погрижи за създаването на обекта за полето **name** (т.е. ще извика конструктора на класа **String**, който ще свърши работата по създаването на низа):



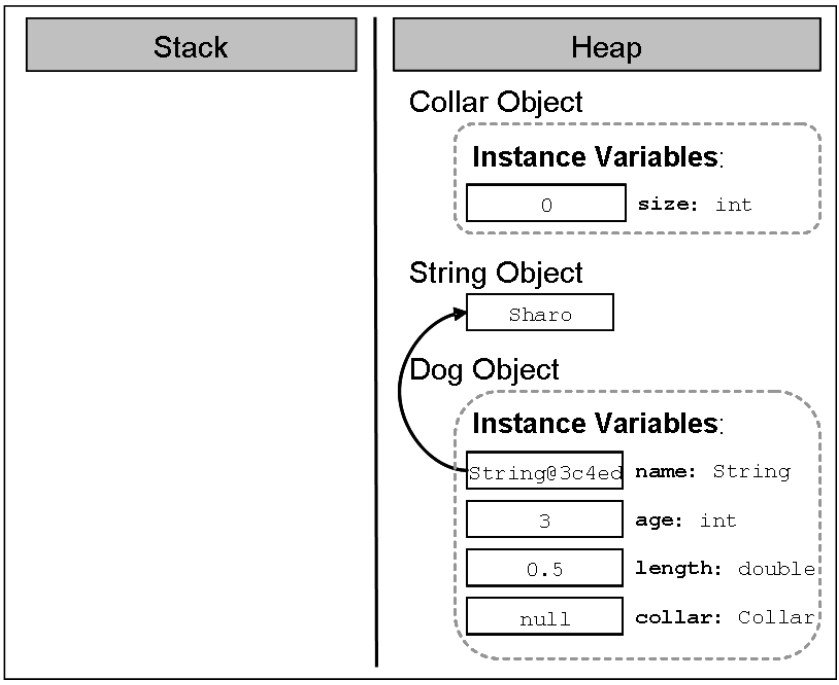
След това нашия конструктор ще запази референция към новия низ в полето **name**:



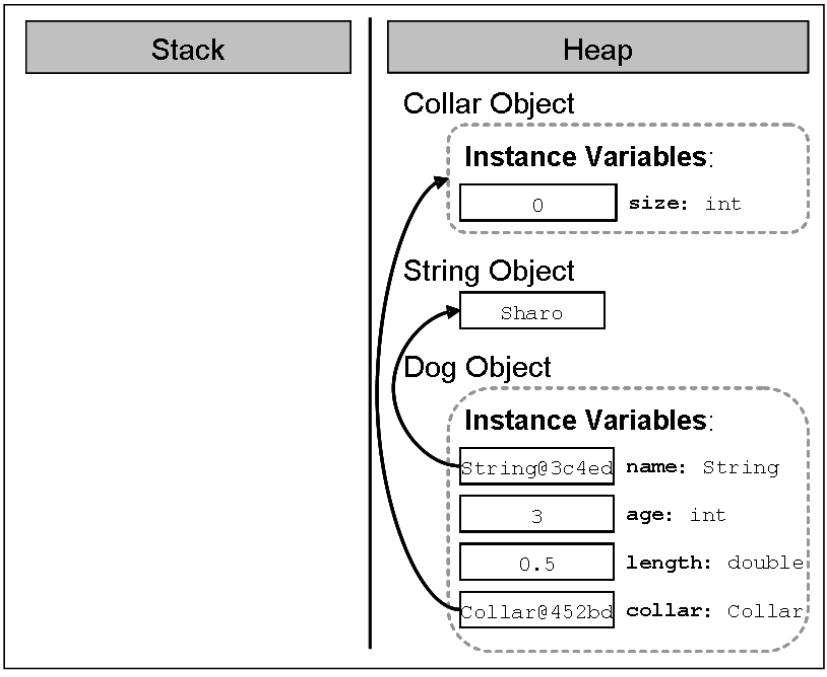
След това идва ред на създаването на обекта от тип **Collar**. Нашият конструктор (на класа **Dog**), извиква конструктора на класа **Collar**, който заделя памет за новия обект:



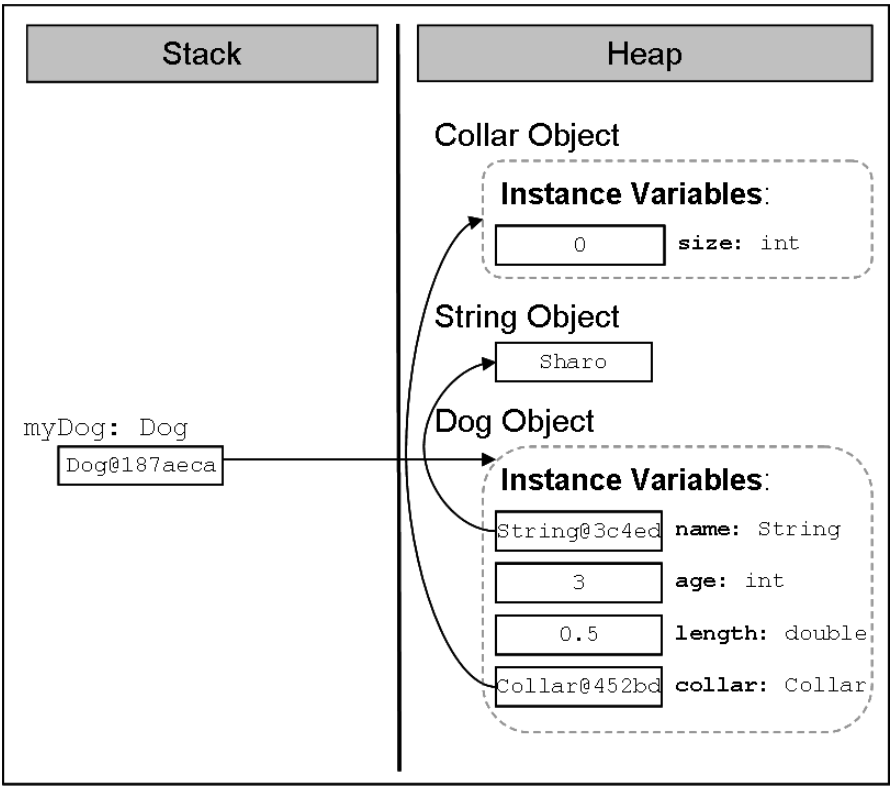
След това я инициализира с подразбиращата се стойност за съответния тип:



След това референцията към новосъздадения обект, която конструкторът на класа **`Collar`** връща като резултат от изпълнението си, се записва в полето **`collar`**:



Накрая референцията към новия обект от тип Dog се присвоява на локалната променлива **myDog** в метода **main()**:



Помним, че локалните променливи винаги се съхраняват в областта от оперативната памет, наречена стек, а обектите – в частта, наречена хийп.

Последователност на инициализиране на полетата на класа

За да няма объркания, нека отбележим последователността, в която се инициализират полетата на един клас, независимо от това дали сме им дали стойност по време на декларация и/или сме ги инициализирали в конструктора.

Първо се заделя памет за съответното поле в хийпа и тази памет се инициализира със стойността по подразбиране на типа на полето. Например, нека разгледаме отново нашия клас **Dog**:

Dog.java

```
public class Dog {
   String name;

   public Dog() {
```

```
        System.out.printf("this.name has value of: %s\n",
           this.name);
        // ... No other code here ...
    }
    // ... Rest of the class body ...
}
```

При опит да създадем нов обект от тип нашия клас, в конзолата ще бъде отпечатано съответно:

```
this.name has value of: null
```

Втората стъпка на виртуалната машина, след инициализирането на полетата със стойността по подразбиране за съответния тип е, ако е зададена стойност при декларацията на полето, тя да му се присвои.

С други думи, ако променим реда от класа **Dog**, на който декларираме полето **name**, го променим по следния начин:

```
String name = "Walcho";
```

Полето първоначално ще бъде инициализирано със стойност **null** и след това ще му бъде присвоена стойността **Walcho**.

Съответно, при всяко създаване на обект от нашия клас:

```
public static void main(String[] args) {
    Dog dog = new Dog();
}
```

Ще бъде извеждано:

```
this.name has value of: Walcho
```

Едва след тези две стъпки на инициализация на полетата на класа (инициализиране със стойностите по подразбиране и евентуално стойността зададена от програмиста по време на декларация на полето), се извиква конструкторът на класа. Едва тогава, полетата получават стойностите, с които са им дадени в тялото на конструктора.

Модификатор final на полета и стойности по подразбиране

Ако се опитаме да декларираме поле с модификатор **final** и не го инициализираме на нито едно място в класа, то това поле няма да бъде инициализирано със стойността по подразбиране. Затова, даването на стойност на това поле, трябва да стане на реда на неговото деклариране. В противен случай, компилаторът ще изведе грешка.

Например, ако се опитаме да компилираме класа **FinalModifierTest**, като не инициализираме полето по време на неговото деклариране:

FinalModifierTest.java

```
class FinalModifierTest {
   final int age; // There is no initializing value

   public static void main(String[] args) {
      FinalModifierTest instance = new FinalModifierTest();
   }
}
```

Ще получим следното съобщение за грешка:

```
The blank final field age may not have been initialized
```

За да избегнем това, трябва или да дадем стойност на полето **age** на реда на неговата декларация, или да декларираме конструктор и да го инициализираме в него. Например:

```
FinalModifierTest() {
   age = 3;
}
```

Сега класът ни се компилира без проблеми.

Деклариране на конструктор с параметри

В предната секция, видяхме как можем да дадем стойности на полетата, различни от стойностите по подразбиране. Много често обаче, по време на декларирането на конструктора, не знаем какви стойности ще приемат различните полета. За да се справим с този проблем, по подобие на методите с параметри, нужната информация, която трябва за работата на конструктора, му се подава чрез списъка с параметри. Например:

```
public Dog(String dogName, int dogAge, double dogLength) {
   name = dogName;
   age = dogAge;
   length = dogLength;
   collar = new Collar();
}
```

Съответно извикването на конструктор с параметри, става по същия начин както извикването на метод с параметри – нужните стойности ги подаваме в списък, чийто елементи са разделени със запетайки:

```
public static void main(String[] args) {

   Dog myDog = new Dog("Bobi", 2, 0.4); // Passing parameters

   System.out.println("My dog " + myDog.getName() +
      " is " + myDog.age+ " year(s) old. " +
      " and it has length: " + myDog.length  + " m");
}
```

Резултатът от изпълнението на този **main()** метод е следния:

```
My dog Bobi is 2 year(s) old. It has length: 0.4 m
```

Трябва да знаем, че нямаме ограничение за броя на конструкторите, които можем да създадем. Единственото условие е те да се различават по сигнатурата си (какво е сигнатура обяснихме в главата "Методи").

Област на действие на параметрите на конструктора

По аналогия на областта на действие на променливите в списъка с параметри на един метод, променливите в списъка с параметри на един конструктор имат област на действие от отварящата скоба на конструктора до затварящата такава, т.е. в цялото тяло на конструктора.

Много често, когато декларираме конструктор с параметри, е възможно да именуваме променливите от списъка му с параметри, със същите имена, като имената на полетата, които ще бъдат инициализирани. Нека за пример вземем отново конструктора, който декларирахме в предходната секция:

```
public Dog(String name, int age, double length) {
   name = name;
   age = age;
   length = length;
   collar = new Collar();
}
```

Нека компилираме и изпълним съответно **main()** метода, който също използвахме в предходната секция. Ето какъв е резултатът от изпълнението му:

```
My dog null is 0 year(s) old. It has length: 0.0 m
```

Странен резултат, нали? Всъщност се оказва, че не е толкова странен. Обяснението е следното – областта, в която действат променливите от списъка с параметри на конструктора, припокрива областта на действие на полетата, които имат същите имена, в конструктора. По този начин не даваме никаква стойност на полетата, тъй като на практика ние не ги достъпваме. Например, вместо на полето **age**, ние присвояваме стойността на променливата **age** на самата нея:

```
age = age;
```

Както видяхме в секцията "Припокриване на област на действие на полета и локални променливи", за да избегнем това разминаване, трябва да достъпим полето, на което искаме да присвоим стойност, но чието име съвпада с името на променлива от списъка с параметри, използвайки ключовата дума **this**:

```
public Dog(String name, int age, double length) {
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = new Collar();
}
```

Сега, ако изпълним отново **main()** метода:

```
public static void main(String[] args) {

   Dog myDog = new Dog("Bobi", 2, 0.4);

   System.out.println("My dog " + myDog.getName() +
      " is " + myDog.age+ " year(s) old. " +
      " and it has length: " + myDog.length  + " m");
}
```

Резултатът ще бъде точно какъвто очакваме да бъде:

```
My dog Bobi is 2 year(s) old. It has length: 0.4 m
```

Конструктор с променлив брой аргументи

Подобно на методите с променлив брой аргументи, които разгледахме в главата "Методи", конструкторите също могат да бъдат декларирани с параметър за променлив брой аргументи. Правилата за декларация и извикване на конструктори с променлив брой аргументи са същите, каквито описахме за декларацията и извикването при методи:

- Когато декларираме конструктор с променлив брой аргументи, трябва да декларираме типа на аргументите, които ще бъдат подавани на метода, следван от три точки, след което името на масива, в който ще се съхраняват тези аргументи. Например за целочислени аргументи – **int... numbers.**
- Позволено е, конструкторът с променлив брой параметри да има и други параметри в списъка си от параметри.

- Параметърът за променлив брой аргументи трябва да е последен в списъка от параметри на конструктора.

Нека разгледаме примерна декларация на конструктор на клас, който описва лекция:

```
public Lecture(String subject, String... studentsNames) {
   // ... Initialization of the instance variables ...
}
```

Първият параметър в декларацията е името на предмета, по който е лекцията, а следващия параметър е за променлив брой аргументи – имената на студентите. Ето как би изглеждало примерното създаването на обект от този клас:

```
Lecture lecture =
   new Lecture("Biology", "Pencho", "Mincho", "Stancho");
```

Съответно, като първи параметър сме подали името на предмета – **"Biology"**, а всички оставащи аргументи – имената на присъстващите студенти.

Варианти на конструкторите (overloading)

Както видяхме, можем да декларираме конструктори с параметри. Това ни дава възможност да декларираме конструктори с различна сигнатура (брой и подредба на параметрите), с цел да предоставим удобство на тези, които ще създават обекти от нашия клас. Създаването на конструктори с различна сигнатура се нарича създаване на **варианти на конструкторите (constructors" overloading)**.

Нека вземем за пример класа **Dog**. Можем да декларираме различни конструктори:

```
public Dog() {                              // NO parameters
   this.name = "Sharo ";
   this.age = 1;
   this.length = 0.3;
   this.collar = new Collar();
}

public Dog(String name) {                 // One parameter
   this.name = name;
   this.age = 1;
   this.length = 0.3;
   this.collar = new Collar();
}
```

```
public Dog(String name, int age) {     // Two parameters

   this.name = name;
   this.age = age;
   this.length = 0.3;
   this.collar = new Collar();
}

public Dog(String name, int age,       // Three parameters
             double length) {
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = new Collar();
}

public Dog(String name, int age,       // Four parameters
             double length,
             Collar collar) {
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = collar;
}
```

Преизползване на конструкторите

В последния пример, който дадохме, видяхме, че в зависимост от нуждите за създаване на обекти от нашия клас, може да декларираме различни варианти на конструктори. Но също така забелязваме, че голяма част от кода на тези конструктори се повтаря. Това ни кара да се замислим, дали няма начин един конструктор, който вече извършва дадена инициализация, не може да бъде преизползван от другите, които правят същата инициализация. От друга страна, в началото на главата казахме, че един конструктор не може да бъде извикан както се извикват методите, а само чрез ключовата дума **new**.

В Java, съществува механизъм, чрез който един конструктор да извиква конструктор деклариран в същия клас:

```
this([<parameters_list>])
```

Извикването става с помощта на ключовата дума **this**, следвана от кръгли скоби. Ако конструкторът, който искаме да извикаме е с параметри, в скобите трябва да добавим списък от параметри, които да му подадем.

Ето как би изглеждал кодът от предната секция, в който вместо да повтаряме инициализацията на всяко едно от полетата, извикваме конструктори, декларирани в същия клас:

```
public Dog() {
   this("Sharo");                                  // Constructor call
}

public Dog(String name) {
   this(name, 1);                                // Constructor call
}

public Dog(String name, int age) {
   this(name, age, 0.3);                         // Constructor call
}

public Dog(String name, int age, double length) {
   this(name, age, length, new Collar()); // Constructor call
}

public Dog(String name, int age,  double length, Collar collar) {
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = collar;
}
```

Преди свободно да декларираме конструктори, които извикват други конструктори в класа, трябва да знаем, че ако в един конструктор викаме друг конструктор, то това трябва да стане на първия ред. Например, ако вземем конструктора с три параметъра по-горе и решим да направим следното:

```
public Dog(String name, int age, double length) {
   Collar collar = new Collar();
   this(name, age, length, collar);     // NOT on the first line
}
```

Компилатора ще изведе следното съобщение за грешка:

```
Constructor call must be the first statement in a constructor.
```

Нека отбележим още веднъж:

Ако в един клас, един конструктор вика друг конструктор от същия клас, то извикваният конструктор трябва да е на

	първия ред в извикващия конструктор. В противен случай, класът няма да се компилира.

Конструктор по подразбиране (implicit constructor)

Нека разгледаме следния въпрос – какво става, ако не декларираме конструктор в нашия клас? Как ще създадем обекти от този тип?

Когато не декларираме нито един конструктор, компилаторът ще създаде един за нас и той ще се използва при създаването на обекти от типа на нашия клас. Този конструктор се нарича **конструктор по подразбиране** (**implicit constructor**).

Когато не дефинираме нито един конструктор в даден клас, компилаторът ще създаде един, наречен конструктор по подразбиране.

Например, декларираме класа **Collar**, без да декларираме никакъв конструктор в него:

Collar.java

```
public class Collar {

   private int size;

   public int getSize() {
      return this.size;
   }
}
```

Въпреки това ще можем да създадем обекти от тип, този клас, по следния начин:

```
Collar collar = new Collar();
```

Конструкторът по подразбиране изглежда по следния начин:

```
<class_access_level> <class_name>()
```

Трябва да знаем, че конструкторът по подразбиране винаги носи името на класа **<class_name>**, винаги списъкът му с параметри е празен и винаги нивото му на достъп съвпада с нивото на достъп на класа **<class_access_level>**.

Конструкторът по подразбиране е винаги без параметри.

За да се уверим, че конструктора по подразбиране винаги е без параметри, нека направим опит да извикаме подразбиращия се конструктор, като му подадем параметри:

```
Collar collar = new Collar(5);
```

Компилаторът ще изведе следното съобщение за грешка:

```
The constructor Collar(int) is undefined.
```

Работа на конструктора по подразбиране

Както се досещаме, единственото, което конструктора по подразбиране ще направи при създаването на обекти от нашия клас, е да задели памет за полетата на класа ни (които не са статични) и да ги инициализира с подразбиращите се стойности. Например, ако в класа **Collar** не сме декларирали нито един конструктор и създадем обект от него и се опитаме да отпечатаме стойността в полето **size**:

```
public static void main(String[] args) {
   Collar collar = new Collar();
   System.out.println("Collar's size is: " + collar.getSize());
}
```

Резултатът ще бъде:

```
Collar's size is: 0
```

Виждаме, че стойността, която е запазена в полето **size** на обекта **collar**, е точно стойността по подразбиране.

Разлика между конструктор по подразбиране и конструктор без параметри

Трябва да знаем, че ако декларираме поне един конструктор в един клас, тогава компилаторът няма да създаде конструктор по подразбиране.

За да проверим това, нека разгледаме следния пример:

```
public Collar(int size) {
   this();
   this.size = size;
}
```

Нека това е единственият конструктор на класа **Collar**. В него се опитваме да извикаме конструктор без параметри, надявайки се, че компилаторът ще е създал конструктор по подразбиране за нас (който знаем, че е без параметри). След като се опитаме да компилираме, ще разберем, че това, което се опитваме да направим, е невъзможно. След като сме декларирали дори един единствен конструктор, компилаторът няма да създаде конструктор по подразбиране за нас:

```
The constructor Collar() is undefined.
```

Преди да приключим със секцията за конструкторите, нека кажем нещо много важно:

Въпреки че конструкторът по подразбиране и този, без параметри, си приличат по сигнатура, те са напълно различни.

Конструкторът по подразбиране се създава от компилатора, ако не декларираме нито един конструктор в нашия клас, а конструкторът без параметри го декларираме ние. Освен това конструкторът по подразбиране винаги ще има нивото на достъп, което има класа. Нивото на достъп на конструктора без параметри зависи отново от нас – ние го определяме.

Модификатор final – особености

До момента видяхме употребата на модификатора **final**, при декларацията на полетата на класа. По-късно в тази глава, в секцията "Константи (constants)", ще видим как с негова помощ ще декларираме константи на класа.

Деклариране на локални променливи с модификатор final

Сега ще разгледаме случая, когато той може да бъде използван при декларацията на локални променливи.

От главата "Методи" помним, че когато декларираме променлива в тялото на един метод, я наричаме "локална променлива" (local variable) за метода. Съответно, ако декларираме една променлива в тялото на конструктор, тя също се нарича локална.

Трябва да знаем, че една локална променлива може да бъде декларирана с модификатор **final**.

Както вече се досещаме, това ще означава, че веднъж инициализирана, на тази променлива няма да може да бъде присвоявана друга стойност. Нека въпреки всичко опитаме:

```
public static void main(String[] args) {
```

```
   final int number = 3;   // Declaring final local variable

   number = 4;             // Unsuccessfull attempt to modify it
}
```

Съответния "отговор", който получаваме от компилатора за последния ред на метода **main()**, е следният:

```
The final local variable number cannot be assigned. It must be blank
and not using a compound assignment.
```

От грешката е ясно, че когато инициализираме една локална променлива веднъж, не можем да го направим втори път. Съответно, за да решим проблема, трябва да махнем реда, на който се опитваме да модифицираме нашата локална променлива, или да махнем **final** от декларацията й.

Това, че една локална променлива може да бъде декларирана с модификатор **final**, не променя това, което знаем от главата "Методи". То е, че преди да бъдат използвани локалните променливи, те трябва да бъдат винаги инициализирани.

Ако в нашия пример се опитаме да декларираме нашата променлива **number** и да пресметнем квадрата й в последствие:

```
public static void main(String[] args) {

   final int number;

   int square = number*number; // Uninitialized local variable...
}
```

Ще получим познатата грешка:

```
The local variable number may not have been initialized.
```

Съответно, решението е да инициализираме променливата по време на нейната декларация.

Локални променливи от референтен тип и модификатор final

До момента винаги разглеждахме примери, в които, полето или локалната променлива, които се декларират с модификатор **final**, са от примитивен тип. Това беше целенасочено, тъй като не знаехме същината на обектите, как те се представят в паметта и т.н. Сега обаче, след като знаем всичко това, ще разгледаме и случаят, когато декларираме с модификатор **final** поле или локална променлива от референтен тип.

Принципно, поведението на една променлива или поле от референтен тип, декларирани с модификатор **final** е същото както и при локална

променлива или поле от примитивен тип – веднъж инициализирани, те не могат да получат друга стойност.

```
public static void main(String[] args) {
   final Dog dog = new Dog("Walcho");
   dog = new Dog("Sharo");
}
```

Разбира се това, което се опитваме да направим в кода е некоректно и компилаторът ще изведе познатото ни съобщение за грешка:

```
The final local variable dog cannot be assigned. It must be blank and
not using a compound assignment
```

Тънкостта тук е, че на една локална променлива от референтен тип, след първоначалната инициализация, не можем да присвоим референция към друг обект, но можем да модифицираме стойностите на полетата на обекта (стига самият обект да позволява това разбира се).

Стойността (т. е. референцията), която се пази в една final променлива (или поле) от референтен тип, не може да бъде модифицирана (т.е. да й бъде присвоена референция към друг обект), но е възможно, обектът, към който "сочи" въпросната референция да бъде модифициран.

Нека разгледаме горния пример:

```
public static void main(String[] args) {
   final Dog dog = new Dog("Walcho");
   System.out.printf("My dog's name is: %s\n", dog.getName());

   dog.setName("Sharo");
   System.out.printf("My dog's name is: %s\n", dog.getName());
}
```

Съответно изходът от изпълнението на този код е следният:

```
My dog's name is: Walcho
My dog's name is: Sharo
```

Както виждаме, не променяме стойността на променливата **dog**, но променяме стойността на полето **name** на обекта от тип **Dog**.

Описаното поведение е същото, когато говорим и за поле на клас, декларирано с модификатор **final**.

Деклариране на методи и конструктори с параметри, които имат модификатор final

Когато декларираме метод или конструктор с параметри, параметрите му в списъка с параметри могат да бъдат декларирани с модификатор **`final`**. Това става, като в с декларацията на съответния параметър, **пред типа** на параметъра поставим ключовата дума **`final`**:

```
public void doSth(final int arg) {
   // Do something...
}
```

Както знаем, параметрите от списъка с параметри на метод или конструктор, технически са просто локални променливи. Следователно, всичко, което казахме до тук за локалните променливи декларирани с модификатор **`final`**, е в сила и за параметри декларирани с този модификатор, независимо дали са от примитивен или референтен тип.

Може би изниква въпроса, защо би ни било нужно да декларираме параметър на метод или още повече на конструктор с такъв модификатор? Отговорът е, че модификаторът **`final`**, ни осигурява защита. Той ни предпазва някой да не "подмени" обектът, към който сочи нашия параметър с чужд обект. С други думи, ако един параметър в декларацията на един метод не е деклариран с модификатор **`final`**, това позволява на произволно място в тялото на метода, на въпросния параметър да се присвои референция към друг обект. По този начин нашият метод няма да работи както се очаква и така ще се наруши сигурността на цялата програма.

Може би всичко това звучи странно, но всичко ще се изясни, след като се запознаем с главата "Принципи на обектно-ориентираното програмиране".

Свойства (properties)

В някои езици за обектно-ориентирано програмиране (например Delphi, Free Pascal, Visual Basic, D, Python), **свойство (property)** се нарича специален вид елемент на класа, който е нещо средно между поле и метод. Характерното е, че достъпът и модификацията на свойствата на класа се осъществява единствено чрез специален вид методи.

Свойствата в Java

Въпреки, че Java е обектно-ориентиран език, в нейната спецификация няма елемент от класа, който да съответства на идеята за свойство. От друга страна, тъй като използването свойства е доказано добра практика и важна част от концепциите на обектно-ориентираното програмиране, в тази секция ще разгледаме, как свойствата могат да бъдат реализирани в един Java клас. Това става чрез деклариране на два метода – един за достъп

(четене) и един за модификация (записване) на стойността на съответното свойство.

Нека разгледаме един пример. Нека имаме отново клас **Dog**, който описва куче. Характерно свойство за едно куче е, например, цвета му (colour). Достъпът до свойството цвят на едно куче може да осъществим по следния начин:

```
// Getting property
String colourName = dogInstance.getColour();

// Setting property
dogInstance.setColour("black");
```

Свойства – капсулация на достъпа до полетата

Основната цел на свойствата е да осигуряват капсулация на класа, в който са декларирани.

Капсулацията (encapsulation) наричаме скриването на физическото представяне на данните в един клас, така че, ако в последствие променим това представяне, това да не рефлектира върху останалите класове, които използват нашия клас.

Чрез синтаксиса на Java, най-често това става, като декларираме полета (физическото представяне на данните) с възможно най-ограничено ниво на видимост (най-често с модификатор **private)** и декларираме достъпът до тези полета (четене и модифициране) да може да се осъществява единствено чрез специални методи.

Капсулация – пример

За да онагледим какво представлява капсулацията, която предоставят свойствата на един клас, както и самите свойства, ще разгледаме един пример.

Нека имаме клас, който представя точка от двумерното пространство със свойства – координатите (x, y). Ето как би изглеждал той, ако декларираме всяка една от координатите, като поле:

Point.java

```
public class Point {

   private double x;
   private double y;

   public Point(int x, int y) {
      this.x = x;
```

```
      this.y = y;
   }

   public double getX() {
      return x;
   }

   public void setX(double x) {
      this.x = x;
   }

   public double getY() {
      return y;
   }

   public void setY(double y) {
      this.y = y;
   }
}
```

Както виждаме, полетата на обектите от нашия клас (т.е. координатите на точките), не могат да бъдат достъпвани чрез точкова нотация. Ако създадем обект от клас **Point**, ние можем да модифицираме и четем свойствата на точката, единствено чрез методите за достъп до тях:

PointTest.java

```
public class PointTest {

   public static void main(String[] args) {
      Point myPoint = new Point(2, 3);

      double myPointXCoordinate = myPoint.getX(); // Access
      double myPointYCoordinate = myPoint.getY(); // Access

      System.out.println("The X coordinate is: " +
         myPointXCoordinate);
      System.out.println("The Y coordinate is: " +
         myPointYCoordinate);
   }
}
```

Резултатът от изпълнението на този **main()** метод ще бъде:

```
The X coordinate is: 2.0
The Y coordinate is: 3.0
```

Ако обаче решим, да променим вътрешното представяне на свойствата на точката, например вместо две полета, ги декларираме като едномерен масив с два елемента:

Point.java

```
public class Point {

   private double[] coordinates;

   public Point(int x, int y) {
      coordinates = new double[2];

      // Initializing the x coordinate
      coordinates[0] = x;

      // Initializing the y coordinate
      coordinates[1] = y;
   }

   public double getX() {
      return coordinates[0];
   }

   public void setX(double x) {
      coordinates[0] = x;
   }

   public double getY() {
      return coordinates[1];
   }

   public void setY(double y) {
      coordinates[1] = y;
   }
}
```

Резултатът от изпълнението на **main()** метода няма да се промени и резултатът ще бъде същия, без да променяме дори символ в кода на класа **PointTest**.

Точно това е добър пример за добра капсулация на данните на един обект, която предоставят свойствата – скриваме вътрешното представяне на информацията, като декларираме методи за достъп до него и ако в последствие настъпи промяна в него, това няма да рефлектира върху другите класове, които използват нашия клас, тъй като те ползват само методите му и не знаят как е представена информацията "зад кулисите".

Конвенция за свойствата в Java

За да декларираме едно свойство в Java, трябва да декларираме методи за достъп (четене и промяна) на съответното свойство, които имат строго определени правила за именуване. Също така трябва да решим по какъв начин ще съхраняваме информацията за това свойство в класа.

За всяко едно от тези условия има конвенция (която е част от JavaBeans спецификацията). Сега ще дадем подробна информация за всяко едно от тях.

Физическо представяне на свойствата в класа

Както видяхме по-горе, свойствата могат да имат различно представяне в един клас. В нашия пример, свойствата на класа **`Point`**, първоначално бяха представени като две полета, след това като чрез едно поле-масив.

Ако обаче решим, вместо да пазим информацията за свойствата на точката в полета, можем да я запазим във файл или база данни и всеки път, когато се наложи да достъпваме съответното свойство, можем да четем/пишем от файла или базата, вместо да достъпваме полетата на класа. Тъй като свойствата се достъпват чрез методи, които ще разгледаме след малко, за класовете, които ще използват нашия клас, това как се съхранява информацията няма да има значение (заради добра капсулация!).

В най-честия случай обаче, информацията за свойствата на класа се пази в поле на класа, което както казахме по-горе, има възможно най-стриктно ниво на видимост. Най-често, нивото на достъп на свойствата на полетата е **`private`**.

Принципно няма значение по какъв начин физически ще бъде пазена информацията за свойствата в един Java клас, но обикновено това става чрез поле на класа с максимално рестриктирано ниво на достъп.

Представяне на свойство без декларация на поле

Нека разгледаме един пример, в който свойството не се пази нито в поле, нито някъде другаде, а се преизчислява.

Нека имаме клас **`Rectangle`**, който представя геометричната фигура правоъгълник. Съответно този клас има две полета – за ширина **`width`** и дължина **`height`**. Нека също така нашия клас има едно свойство – лице, **`area`**. Тъй като винаги чрез дължината и ширината на правоъгълника можем да намерим стойността на свойството лице, не е нужно да имаме отделно поле в класа, за да пазим тази стойност. По тази причина, можем да си декларираме просто един метод за получаване на лицето, в който пресмятаме формулата за лице на правоъгълник:

Rectangle.java

```
public class Rectangle {
   private float width;
   private float height;

   public Rectangle(float width, float height) {
      this.width = width;
      this.height = height;
   }

   // Obtaining the value of the property area
   public float getArea() {
      return this.width * this.height;
   }
}
```

Както ще видим след малко не е задължително едно свойство да има едновременно метод за модификация и за четене на стойността. Затова е напълно легално, ако декларираме само метод за чете на свойството **area** на правоъгълника – няма смисъл от метод, който модифицира стойността на лицето на един правоъгълник, тъй като то е винаги едно и също при определена дължина на страните.

Метод за четене на стойността на свойство

При декларацията на метод за четене на стойността на едно свойство трябва да се спазват определени правила:

```
public <property_type> get<property_name>()
```

Методът трябва винаги да е деклариран с ниво на видимост **public** и трябва да има тип на връщана стойност, същия, като типа на свойството - **<property_type>**. Освен това, името на метода, трябва да се конструира по специално правило – той винаги започва с **get**, след което следва името на свойството, започвайки с главна буква, спазвайки правилото **camelCase**. Например, ако имаме свойство **height**, то името на метода за четене на стойността на това свойство, трябва да бъде **getHeight**.

Ето няколко примера за правилно декларирани методи за четене на свойство:

```
public int getMyValue()      // myValue property
public String getColour()    // colour property
public double getX()         // x-coordinate property
```

Правилото за именуване на метод за четене на стойност на свойство има една добавка. Когато свойството е от тип **boolean** или **Boolean**, тогава освен с **get**, името на метода може да започва с **is**. Например:

```
public boolean isStopped()     // stopped property
public boolean isMyStatus()  // myStatus property
```

Метод за промяна на стойността на свойство

По подобие на метода за четене на стойността на едно свойство, когато декларираме метод за промяна стойността на едно свойство, трябва да се съобразяваме с дадени правила:

```
public void set<property_name>(<property_type> parameter_name)
```

Методът, който модифицира едно свойство, винаги трябва да е с **public** модификатор за достъп и винаги да е с тип на връщана стойност **void**.

Също така, името на метода трябва да започва винаги със **set**, следвано от името на свойството, като първата буква от името на свойството е главна (т.е. отново спазваме правилото **camelCase**). Например, ако имаме свойство **width**, методът за промяна на това свойство трябва да се именува – **setWidth**.

Последното условие, на което трябва да отговаря един метод за промяна стойността на свойство е, че той трябва да има списък с параметри, който има точно един параметър. Този параметър трябва да е със същия тип, какъвто е типът на свойството – **<property_type>**.

Нека разгледаме няколко примера за правилно декларирани методи за промяна на свойство:

```
public void setMyValue(int myValue) // myValue :: int, property
public void setColor(String colourName)  // color :: String,
                                           // property
public void setX(double x) // x-coordinate :: double, property
```

За разлика от методите за четене на свойство, когато декларираме метод за промяна на свойство, което е от тип **boolean**, няма промяна в префикса **set** от правилото за създаване името на този метод:

```
public void setStopped(boolean stopped)     // stopped :: boolean,
                                              // property
```

Видове свойства

Преди да приключим трябва да кажем още нещо за свойствата в един клас. В зависимост от особеностите им, можем да декларираме свойствата по следния начин:

- Само за четене, т.е. тези свойства имат само **get**-метод, както в примера с лицето на правоъгълник.
- Само за модифициране, т.е. тези свойства имат само set-метод, но не и метод за четене на стойността на свойството.
- И най-честият случай е, когато свойството може да има метод както за четене, така и за промяна на стойността.

Статични членове на класа (static members)

Когато един елемент на класа е деклариран с модификатор **static**, ние го наричаме статичен. Тъй като статични могат да бъдат само полетата и методите на класа, когато говорим за статични членове (елементи) на класа ще визираме точно тях.

За какво се използват статичните елементи?

Преди да разберем принципа, на който работят статичните елементи на класа, нека разгледаме причините, поради които се налага използването им.

Метод за сбор на две числа

Нека си представим, че имаме клас, в който един метод винаги работи по един и същ начин. Например, нека неговата задача е да събира две числа, подадени в списъка му от параметри и да връща резултата от сбора им. Виждаме, че няма да има никакво значение кой обект от този клас ще изпълни този метод, тъй като той винаги ще се държи по един и същ начин – ще събира две числа, независими от извикващия обект.

С други думи, поведението на метода не зависи от състоянието на обекта (стойностите в полетата на обекта). Тогава защо е нужно да създаваме обект, за да изпълним този метод, при положение, че метода не зависи от никой от обектите от този клас? Защо просто не накараме класа да изпълни този метод?

Брояч на инстанциите от даден клас

Нека разгледаме и друг сценарий. Да кажем, че искаме да пазим в програмата ни текущия брой на обектите, които са били създадени от даден клас. Как ще съхраним тази променлива, която ще пази броя на създадените обекти?

Както знаем, няма да е възможно да я пазим като поле на класа, тъй като при всяко създаване на обект, ще се създава ново копие на това поле за всеки обект, и то ще бъде инициализирано със стойността по подразбиране. Всеки обект ще пази свое поле за индикация на броя на обектите и обектите няма да могат да споделят информацията по между си. В следващите подсекции ще разберем как да се справим и с този проблем.

Какво е статичен член?

Формално погледнато, **статичен член (static member)** на класа наричаме поле или метод, който има модификатор **`static`** в декларацията си. Това означава, че полета и методи маркирани като статични, принадлежат на самия клас, а не на някой конкретен обект от дадения клас.

Следователно, когато маркираме поле или метод като статични, можем да ги използваме, без да създаваме нито един обект от дадения клас. Единственото, от което се нуждаем е да имаме достъп (видимост) до класа, за да можем да извикваме статичните методи, или да достъпваме статични полета.

Статичните елементи на класа могат да се използват, без да се създава обект от дадения клас.

От друга страна, ако имаме създадени обекти от дадения клас, тогава статичните полета ще бъдат общи за тях – има само едно копие на статично поле, което се споделя от всички обекти от дадения клас.

Статични полета

Когато създаваме обекти от даден клас, всеки един от тях има различни стойности в полетата си. Например, нека разгледаме отново класа **`Dog`**:

Dog.java

```
public class Dog {

   // Instance variables
   private String name;
   private int age;
}
```

Той има две полета съответно за име – **`name`** и възраст – **`age`**. Във всеки обект, всяко едно от тези полета има собствена стойност, която се съхранява на различно място в паметта за всеки обект.

Понякога обаче, искаме да имаме полета, които са общи за всички обекти от даден клас. За да постигнем това, трябва в декларацията на тези полета да използваме модификатора **`static`**. Както казахме, такива полета се

наричат **статични полета**. В литературата се срещат, също и като **променливи на класа**.

Декларация на статични полета

Статичните полета ги декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **static**:

```
[<access_modifier>] static <field_type> <field_name>
```

Ето как би изглеждало едно поле **dogCount**, което пази информация за броя на създадените обекти от клас **Dog**:

Dog.java

```
public class Dog {

   // Static variable
   static int dogCount;

   // Instance variables
   private String name;
   private int age;
}
```

Статичните полета се създават, когато за първи път се опитаме да създадем обект от класа, на който принадлежат или когато заредим класа в паметта (как става това обаче, е извън обхвата на тази книга и няма да го разглеждаме). След създаването си, по подобие на обикновените полета в класа, те се инициализират с подразбиращата се стойност за типа си.

Инициализация по време на декларация

Трябва да знаем, че статичните полета са асоциирани с класа, вместо с който и да е обект от дадения клас. Това означава, че всички обекти, създадени по описанието на един клас **споделят** статичните полета на класа.

Ако по време на декларация на статичното поле, сме задали стойност за инициализация, тя се присвоява на съответното статично поле. Тази инициализация се изпълнява само веднъж – при създаването на полето, веднага след като приключи присвояването на стойността по подразбиране. При последващо създаване на обекти от този клас, тази инициализация на статичното поле няма да се изпълни, тъй като статичното поле е асоциирано с класа и е независимо от обектите.

В горния пример например, ако добавим инициализация на статичното поле:

```
// Static variable - declaration and initialization
static int dogCount = 0;
```

Тази инициализация ще се извърши при създаването на първия обект от нашия клас или при първия опит да достъпим статичен елемент на класа (повече за това, в следващата секция). Когато извикаме за първи път конструктора на класа **Dog** или питаме да достъпим някое статично поле или статичен метод на класа, описанието на класа **Dog** ще се зареди в паметта. Тогава ще се задели памет за статичните му полета, те ще се инициализират със стойностите им по подразбиране. След това за тези статични полета, които имат инициализация по време на декларацията си (както е в нашия случай с полето **dogCount**), тази инициализация ще се извърши. Едва след тази инициализация ще се създаде първият обект от класа. В последствие обаче, когато създаваме други обекти от същия клас, този процес няма да се повтори, тъй като статичното поле вече съществува и един път създадено, то се споделя между всички обекти в класа.

Достъп до статични полета

За разлика от обикновените (нестатични) полета на класа, статичните, бидейки асоциирани с класа, а не с конкретен обект, могат да бъдат достъпвани, без да бъде създаван обект от дадения клас. Това става като към името на класа, чрез точкова нотация, достъпим името на съответното статично поле:

```
<class_name>.<static_field_name>
```

Например, ако искаме да отпечатаме стойността на статичното поле, което пази броя на създадените обекти от нашия клас **Dog**, това ще стане по следния начин:

```
public static void main(String[] args) {
   // Access to the static variable through class name
   System.out.println("Dog count is now " + Dog.dogCount);
}
```

Съответно, изходът от изпълнението на този **main()** метод е:

```
Dog count is now 0
```

Въпреки, че по принцип статичните полета се достъпват чрез името на класа, те могат да бъдат достъпвани и чрез променлива, която е референция към конкретен обект:

```
<class_instance_variable>.<static_field_name>
```

Ако модифицираме леко последния пример с метода **main()**, ще видим как можем да извлечем стойността на броя създадени обекти, използвайки някоя локална променлива – **dog1**, която съхранява референция към обект от тип **Dog**:

```
public static void main(String[] args) {
   Dog dog1 = new Dog();

   // Accessing the static variable through local variable
   System.out.println("Dog count is now " + dog1.dogCount);
}
```

Разбира се, изходът е идентичен с този по-горе:

```
Dog count is now 0
```

Трябва да знаем обаче, че по-добра практика е да достъпваме статичните елементи на класа, чрез името на класа, вместо чрез някоя променлива, която съхранява референция към обект от този клас, тъй като това прави кода по-четим. Когато използваме променлива, която "сочи" към обект, не става ясно, че полето, което се достъпва чрез нея, е статично.

Модификация на стойностите на статичните полета

Както казахме по-горе, статичните променливи на класа, са споделени от всички обекти и не принадлежат на нито един обект от класа. Съответно, това дава възможност, всеки един от обектите на класа да променя стойностите на статичните полета, като по този начин останалите обекти ще могат да "видят" модифицираната стойност.

Ето защо, например, за да отчетем броя на създадените обекти от клас **Dog**, е удобно да използваме статично поле, което увеличаваме с единица, при всяко извикване на конструктора на класа, т.е. всеки път, когато създаваме обект от нашия клас:

Dog.java

```
public Dog(String name, int age) {
   this.name = name;
   this.age = age;

   dogCount += 1; // Modify the value in the constructor
}
```

Съответно, за да проверим дали това, което написахме е вярно, ще създадем няколко обекта от нашия клас **Dog** и ще отпечатаме броя им. Това ще стане по следния начин:

```
public static void main(String[] args) {
     Dog dog1 = new Dog("Karaman", 1);
     Dog dog2 = new Dog("Bobi", 2);
     Dog dog3 = new Dog("Sharo", 3);

     // Access to the static variable
     System.out.println("Dog count is now " + Dog.dogCount);
   }
```

Съответно изходът от изпълнението на този **main()** метод е:

```
Dog count is now 3
```

Константи (constants)

Преди да приключим с темата за статичните полета, трябва да се запознаем с един особен вид статични полета.

По подобие на константите от математиката, в Java, могат да се създадат полета на класа, които декларирани и инициализирани веднъж, винаги притежават една и съща стойност. Това са константите.

Декларация на константи

Когато декларираме едно поле с модификатори **static** и **final**, това поле наричаме **константа (constant)**:

```
[<access_modifiers>] static final <type> <name>;
```

Момент на инициализация на константата

Знаем, че когато декларираме едно поле с модификатор **final**, това означава, че можем да му дадем стойност само веднъж. От друга страна, знаем, че при деклариране едно статично поле, то винаги се инициализира със стойността по подразбиране. Ето защо, когато декларираме константа, трябва да винаги да й даваме стойност по време на декларацията:

```
[<access_modifiers>] static final <type> <name> = <value>;
```

Например, ако искаме да декларираме като константа числото Пи, познато ни от математиката, това ще стане по следния начин:

```
public static final double PI = 3.141592653589793;
```

Ако не дадем стойност на дадена константа по време на декларацията й, а по-късно, ще получим грешка при компилация. Например, ако в примера с константата **PI**, първо декларираме константата, и по-късно се опитаме да й дадем стойност:

```
public static final double PI;

// ... Some code ...

public void myMethod() {

   // Attempting to initialize the constant PI
   PI = 3.141592653589793;
}
```

Компилаторът ще изведе грешка подобна на следната:

```
The final field PI cannot be assigned.
```

Нека обърнем внимание отново:

Константите на класа трябва да се инициализират в момента на тяхната декларация.

Форматиране на константите

Съгласно конвенцията, константите в Java, винаги се изписват с главни букви. Ако константата е съставена от няколко думи, те се разделят със символа за долна черта (underscore) – "_". Ето няколко примера за константи:

```
// The base of the natural logarithms (approximate value)
public static final double E = 2.718281828459045;

public static final double PI = 3.141592653589793;
public static final char PATH_SEPARATOR = '/';
public static final String BIG_COFFEE = "big";
public static final int MAX_VALUE = 2147483647;
```

Статични методи

По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен.

Декларация на статични методи

Синтактично, това означава, че в декларацията на метода, трябва да добавим ключовата дума **static**:

```
[<access_modifier>] static <return_type> <method_name>()
```

Нека например декларираме метода за събиране на две числа, за който говорихме в началото на тази секция:

```
public static int add(int number1, int number2) {
   return (number1 + number2);
}
```

Достъп до статични методи

Също както и статичните полета, статичните методи могат да бъдат достъпвани както чрез точкова нотация (операторът точка) приложена към името на класа, така и към променлива, която съхранява референция към обект от дадения клас. Например:

```
public static void main(String[] args) {
   // Access trough the class name
   System.out.println(MyMathClass.add(3, 5));

   // Access trough variable of the class type
   MyMathClass myMathVariable = new MyMathClass();
   System.out.println(myMathVariable.add(3, 5));
}
```

Разбира се по-препоръчително е, както при статичните полета, достъпът до статични методи да става чрез името на класа.

Достъп между статични и нестатични елементи на класа

В повечето случаи статичните методи се използват за достъпване на статични полета. Например, когато искаме да декларираме метод, който да връща броя на създадените обекти от класа **Dog**, той ще е статичен:

```
public static int getDogCount() {
   return dogCount;
}
```

Но когато разглеждаме как статични и нестатични методи и полета могат да се достъпват, не всички комбинации са позволени.

Достъп до нестатичните елементи на класа от нестатичен метод

Нестатичните методи могат да достъпват нестатичните полета и други нестатични методи на класа. Например, в класа **Dog** можем да декларираме метод **getInfo()**, който извежда информация за нашето куче:

Dog.java

```
public class Dog {

   // Static variable
   static int dogCount;

   // Instance variables
   private String name;
   private int age;

   public Dog(String name, int age) {
      this.name = name;
      this.age = age;

      dogCount += 1;
   }

   public void bark() {
      System.out.println("wow-wow");
   }

   // Non-static (instance) method
   public void getInfo() {

      // Accessing instance variables – name and age
      System.out.print("Dog's name: " + this.name +
         "; age: " + this.age + "; often says: ");

      // Calling instance method
      this.bark();
   }
}
```

Разбира се, ако създадем обект от класа **Dog** и извикаме неговия **getInfo()** метод:

```
public static void main(String[] args) {
   Dog dog = new Dog("Sharo", 1);
   dog.getInfo();
}
```

Резултатът ще бъде следният:

```
Dog's name: Sharo; age: 1; often says: wow-wow
```

Достъп до статичните елементи на класа от нестатичен метод

От нестатичен метод, можем да достъпваме статични полета и статични методи на класа. Както разбрахме по-рано, това е така, тъй като статичните методи и променливи са обвързани с класа, вместо с конкретен метод и статичните елементи могат да се достъпват от кой да е обект на класа. Например:

Boo.java

```
public class Boo {

   private static String staticVariable = "test";

   public static void doSomething() {
      System.out.println("doSomething() method execution.");
   }

   public void printStaticClassElements() {
      // Accessing static variable from non-static method
      System.out.println("staticVaruable: "
                         + Boo.staticVariable);

      // Accessing static method from non-static method
      Boo.doSomething();
   }

   public static void main(String[] args) {
      Boo booInstance = new Boo();
      booInstance.printStaticClassElements();
   }
}
```

Въпреки че имената на методите, полето и класа са безсмислени, виждаме че от нестатичния метод **printStaticClassElements()**, можем да достъпим стойността на статичното поле **staticVariable**, както и да извикаме статичния метод **doSomething()**. За целта на демонстрацията обаче, това е достатъчно. След компилация и изпълнение, това, което ще бъде изведено от в конзолата ще бъде:

```
staticVaruable: test
doSomething() method execution.
```

Достъп до статичните елементи на класа от статичен метод

От статичен метод можем да извикваме друг статичен метод или статично поле на класа безпроблемно.

Например, нека вземем нашия клас за математически пресмятания. В него имаме декларирана константата Пи (която е вид статично поле). Можем да декларираме статичен метод за намиране дължината на окръжност (формулата за намиране периметър на окръжност е **2πr**, където **r** е радиусът на окръжността), който за пресмятането на периметъра на дадена окръжност, достъпва константата Пи. След това, за да покажем, че статичен метод може да вика друг статичен метод, можем от метода **main()**, който е статичен, да извикаме статичен метод за намиране дължина на окръжност:

MyMathClass.java

```
public class MyMathClass {

   public static final double PI = 3.141592653589793;

   // P = 2 * PI * r
   public static double getCirclePerimeter(double r) {

      // Accessing the static variable PI from static method
      return (2 * PI * r);
   }

   public static void main(String[] args) {
      double radius = 5;

      // Accessing static method from other static method
      double circlePerimeter = getCirclePerimeter(radius);

      System.out.println("Circle with radius " + radius +
         " has perimeter: " + circlePerimeter);
   }
}
```

Кодът се компилира без грешки и при изпълнение извежда следния резултат:

```
Circle with radius 5.0 has perimeter: 31.41592653589793
```

Достъп до нестатичните елементи на класа от статичен метод

Нека разгледаме най-интересния случай от комбинацията от достъпване на статични и нестатични елементи на класа – достъпването на нестатични елементи от статичен метод.

Трябва да знаем, че от статичен метод не могат да бъдат достъпвани нестатични полета, нито да бъдат извиквани нестатични методи. Това е така, защото статичните методи са обвързани с класа, и не "знаят" за нито един обект от класа. Затова, ключовата дума **this** не може да се използва

в статични методи – тя обвързана с конкретна инстанция на класа. При опит за достъпване на нестатични елементи на класа (полета или методи) от статичен клас, винаги ще получаваме грешка при компилация.

Достъп до нестатично поле от статичен метод – пример

Ако в нашия клас **Dog** се опитаме да декларираме статичен метод **getName()**, който връща като резултат стойността на нестатичното поле **name** декларирано в класа:

```
public static String getName() {
   // Accessing non-static variable from static method
   return name; // INVALID
}
```

Съответно компилаторът ще ни отговори със съобщение за грешка, подобно на следното:

```
Cannot make static reference to the non-static field name.
```

Ако въпреки това, се опитаме в метода да достъпим полето чрез ключовата дума **this**:

```
public static String getName() {
   // Accessing non-static variable from static method by this
   return this.name; // INVALID
}
```

Компилаторът отново няма да е доволен и този път ще изведе следното предупреждение, без да успее да компилира класа:

```
Cannot use this in static context.
```

Извикване на нестатичен метод от статичен метод – пример

Сега ще се опитаме да достъпим нестатичен метод от статичен такъв. Нека в нашия клас **Dog** декларираме нестатичен метод **getAge()**, който връща стойността на полето **age**:

```
public int getAge() {
   return this.age;
}
```

Съответно, нека се опитаме от метода **main()**, който декларираме в класа **Dog**, да извикаме този метод без да създаваме обект от нашия клас:

```
public static void main(String[] args) {
```

```
   // Attempting to invoke non-static method from static one
   int someDogAge = getAge(); // INVALID

   System.out.println("Some dog has age of " +
      someDogAge + " years ");
}
```

При опит за компилация ще получим следната грешка:

```
Cannot make a static reference to non-static method getAge() from
type Dog.
```

Резултатът е подобен, ако се опитаме да измамим компилатора, опитвайки се да извикаме метода чрез ключовата дума **this**:

```
public static void main(String[] args) {

   // Attempting to invoke non-static method from static one
   // by this
   int someDogAge = this.getAge(); // INVALID

   System.out.println("Some dog has age of " +
      someDogAge + " years ");
}
```

Съответно, както в случая за достъп до нестатично поле в статичен метод, чрез ключовата дума **this**, компилаторът извежда следното съобщение, без да успее да компилира нашия клас:

```
Cannot use this in static context.
```

От разгледаните примери, можем да направим следния извод:

Нестатичните елементи на класа НЕ могат да бъдат достъпвани от статични методи.

Проблемът с достъпа до нестатични елементи на класа от статичен метод има едно единствено решение – тези нестатични елементи да се достъпват чрез референция към даден обект:

```
public static void main(String[] args) {
   Dog myDog = new Dog("Sharo", 2);

   String myDogName = myDog.name;
   int myDogAge = myDog.getAge();

   System.out.println("My dog \"" + myDogName +
```

```
        "\" has age of " + myDogAge + " years ");
}
```

Съответно този код се компилира и резултатът от изпълнението му е:

```
My dog "Sharo" has age of 2 years
```

Статични свойства на класа

Макар и рядко, понякога е удобно да се декларират и използват свойства не на обекта, а на класа. Те носят същите характеристики като свойствата, свързани с конкретен обект от даден клас, които разгледахме по-горе, но с тази разлика, че статичните свойства се отнасят за класа. Както можем да се досетим, всичко, което е нужно да направим, за да превърнем едно обикновено свойство в статично, е да добавим ключовата дума **static** в декларацията на методите за четене и модификация на съответното свойство.

Съответно, методите за получаване на стойността на статичното свойство се декларират по следния начин:

```
public static <property_type> get<property_name>()
public static <boolean | Boolean> is<property_name>()
```

А методът за модификация на статичното свойство се декларира така:

```
public static void set<property_name>(<property_type> param)
```

Нека разгледаме един пример. Нека имаме клас, който описва някаква система. Ние можем да създаваме много обекти от нея, но моделът на системата има дадена версия и производител, които са общи за всички екземпляри, създадени от този клас:

System.java

```
public class System {

   private static double version = 0.1;
   private static String vendor = "Sun Microsystems";

   // The version property getter:
   public static double getVersion() {
      return version;
   }

   // The version property setter:
   public static void setVersion(double version) {
```

```
      System.version = version;
   }

   // The vendor property getter:
   public static String getVendor() {
      return vendor;
   }

   // The vendor property setter:
   public static void setVendor(String vendor) {
      System.vendor = vendor;
   }

   // ... More (non)static code here ...
}
```

Тук сме избрали да пазим стойността на статичните свойства в статични променливи (което е логично, тъй като те са обвързани само с класа). Свойствата, които разглеждаме са съответно версия (version) и производител (vendor). За всяко едно от тях сме създали статичен метод за достъп до съответното свойство и модификация. Така всички обекти от този клас, ще могат да извлекат текущата версия и производителя на системата, която описва класа. Съответно, ако някой ден бъде направено обновление на версията на системата например стойността стане **0.2**, всеки от обектите, ще получи като резултат новата версия, чрез достъпване на свойството на класа.

Вътрешни, локални и анонимни класове

В Java можем да дефинираме класове вътре в даден друг клас или дори в даден метод. Понякога това може да е много удобно, когато ни трябва клас, който искаме да използваме временно или искаме да скрием от външния свят.

Вътрешни класове

В Java е възможно в един клас да се дефинира друг клас, т.е. класът да е член на клас. Такъв клас наричаме **вътрешен клас** (**inner class**, **nested class**). Нека разгледаме тази възможност с един пример:

OuterClass.java

```
public class OuterClass {
   private String name;

   private OuterClass(String name) {
      this.name = name;
```

```
   }

   private class InnerClass {
      private String name;

      private InnerClass(String name) {
         this.name = name;
      }

      private void printNames() {
         System.out.println("Inner name: " + this.name);
         System.out.println("Outer name: " +
            OuterClass.this.name);
      }
   }

   public static void main(String[] args) {
      OuterClass outerClass = new OuterClass("outer");
      InnerClass innerClass = outerClass.new InnerClass("inner");
      innerClass.printNames();
   }
}
```

В примера външният клас **OuterClass** дефинира в себе си като **private** член класа **InnerClass**. Нестатичните методи на вътрешния клас имат достъп както до собствената си инстанция **this**, така и до инстанцията на външния клас (чрез синтаксиса **OuterClass.this**). При създаването на вътрешния клас на конструктора му се подава **this** референцията на външния клас, защото вътрешният клас не може да съществува без конкретна инстанция на външния. Забележете, че външния клас може да вика свободно **private** методи и конструктори от вътрешния клас.

Ако изпълним горния пример, ще получим следния резултат:

```
Inner name: inner
Outer name: outer
```

Вътрешните класове могат да бъдат декларирани като статични (чрез модификатора **static**). В този случай те могат да съществуват и без външния клас, в който са разположени, но нямат достъп до неговата **this** инстанция.

Локални класове

В Java можем да дефинираме класове и в даден метод. Наричаме ги **локални класове** (**local classes**). Локалните класове са подобни на вътрешните класове, но не могат да бъдат статични. Те имат достъп до член-променливите и методите на външния им клас. Локалните класове

могат да осъществяват достъп и до променливите, декларирани в метода, в който се съдържат, стига тези променливи да са обявени като **final**. Ето един пример:

LocalClassExample.java

```
public class LocalClassExample {
   public static void main(String[] args) {
      final int value = 5;
      class LocalClass {
         void printSomething() {
            System.out.println(value);
         }
      }
      LocalClass localClass = new LocalClass();
      localClass.printSomething();
   }
}
```

Ако изпълним горния пример, ще получим следния резултат:

```
5
```

Локалните класове са достъпни само и единствено в метода, в който са декларирани и нямат модификатори за видимост и не могат да бъдат статични, както всяка една локална променлива.

Анонимни класове

В Java можем да декларираме локален клас без име. Такъв клас се нарича **анонимен клас** (**annonymous class**). Да разгледаме един пример:

AnnonymousClassExample.java

```
public class AnnonymousClassExample {
   public static void main(String[] args) {
      new Object() {
         void printSomething() {
            System.out.println("I am annonymous class.");
         }
      }.printSomething();
   }
}
```

В примера декларираме клас без име (анонимен клас), който наследява класа **java.lang.Object** и добавя към него нов метод **printSomething()**. След това създаваме инстанция на този анонимен клас и му извикваме добавения метод **printSomething()**. За наследяването ще ви разкажем подробно в

главата "Принципи на ООП". За момента приемете, че анонимните класове са локални класове без име, които ползват за основа даден съществуващ клас и му добавят допълнителни методи.

Ако изпълним горния пример, ще получим следния резултат:

```
I am annonymous class.
```

Упражнения

1. Дефинирайте клас **Student**, който съдържа следната информация за студентите: трите имена, курс, специалност, университет, електронна поща и телефонен номер.
2. Декларирайте няколко конструктора за класа **Student**, които имат различни списъци с параметри (за цялостната информация за даден студент или част от нея). Данните, за които няма входна информация да се инициализират съответно с **null** или **0**.
3. Добавете статично поле в класа **Student**, в което се съхранява броя на създадените обекти от този клас.
4. Добавете метод в класа **Student**, който извежда пълна информация за студента.
5. Модифицирайте текущия код на класа **Student** така, че да капсулирате данните в класа чрез свойства.
6. Напишете клас **StudentTest**, който да тества функционалността на класа **Student**.
7. Добавете статичен метод в класа **StudentTest**, който създава няколко обекта от тип **Student** и ги съхранява в статични полета. Създайте статично свойство на класа, което да ги достъпва. Напишете тестова програма, която да извежда информацията за тях в конзолата.
8. Дефинирайте клас, който съдържа информация за мобилен телефон: модел, производител, цена, собственик, характеристики на батерията (модел, idle time и часове разговор /hours talk/) и характеристики на екрана (големина и цветове).
9. Декларирайте няколко конструктора за всеки от създадените класове от предходната задача, които имат различни списъци с параметри (за цялостната информация за даден студент или част от нея). Данните за полетата, които не са известни трябва да се инициализират съответно със стойности с **null** или **0**.
10. Към класа за мобилен телефон от предходните две задачи, добавете статично поле **nokiaN95**, което да съхранява информация за мобилен телефон модел Nokia 95. Добавете метод, в същия клас, който извежда информация за това статично поле.

11. Дефинирайте свойства, за да капсулирате данните в класовете **GSM**, **Battery** и **Display**.

12. Напишете клас **GSMTest**, който тества функционалностите на класа **GSM**. Създайте няколко обекта от дадения клас и ги запазете в масив. Изведете информация за създадените обекти. Изведете информация за статичното поле **nokiaN95**.

13. Създайте клас **Call**, който съдържа информация за разговор, осъществен през мобилен телефон. Той трябва да съдържа информация за датата, времето на започване и продължителността на разговора.

14. Добавете свойство архив с обажданията – **callHistory**, което да пази списък от осъществените разговори.

15. В класа **GSM** добавете методи за добавяне и изтриване на обаждания (**Call**) в архива с обаждания на мобилния телефон. Добавете метод, който изтрива всички обаждания от архива.

16. В класа **GSM** добавете метод, който пресмята общата сума на обажданията (**Call**) от архива с обаждания на телефона (**callHistory**) като нека цената за едно обаждане се подава като параметър на метода.

17. Създайте клас **GSMCallHistoryTest**, с който да се тества функционалността на класа **GSM**, от задача 12, като обект от тип **GSM**. След това, към него добавете няколко обаждания (**Call**). Изведете информация за всяко едно от обажданията. Ако допуснем, че цената за минута разговор е 0.37, пресметнете и отпечатайте общата цена на разговорите. Премахнете най-дългият разговор от архива с обаждания и пресметнете общата цена за всички разговори отново. Най-накрая изтрийте архива с обаждания.

18. Нека е дадена библиотека с книги. Дефинирайте класове съответно за библиотека и книга. Библиотеката трябва да съдържа име и списък от книги. Книгите трябва да съдържат информация за заглавие, автор, издателство, година на издаване и ISBN-номер. В класа, който описва библиотека, добавете методи за добавяне на книга към библиотеката, търсене на книга по предварително зададен автор, извеждане на информация за дадена книга и изтриване на книга от библиотеката.

19. Напишете тестов клас, който създава обект от тип библиотека, добавя няколко книги към него и извежда информация за всяка една от тях. Имплементирайте тестова функционалност, която намира всички книги, чийто автор е Стивън Кинг и ги изтрива. Накрая, отново изведете информация за всяка една от оставащите книги.

20. Дадено ни е училище. В училището имаме класове и ученици. Всеки клас има множество от преподаватели. Всеки преподавател има множество от дисциплини, по които преподава. Учениците имат име и уникален номер в класа. Класовете имат уникален текстов идентификатор. Дисциплините имат име, брой уроци и брой упражнения.

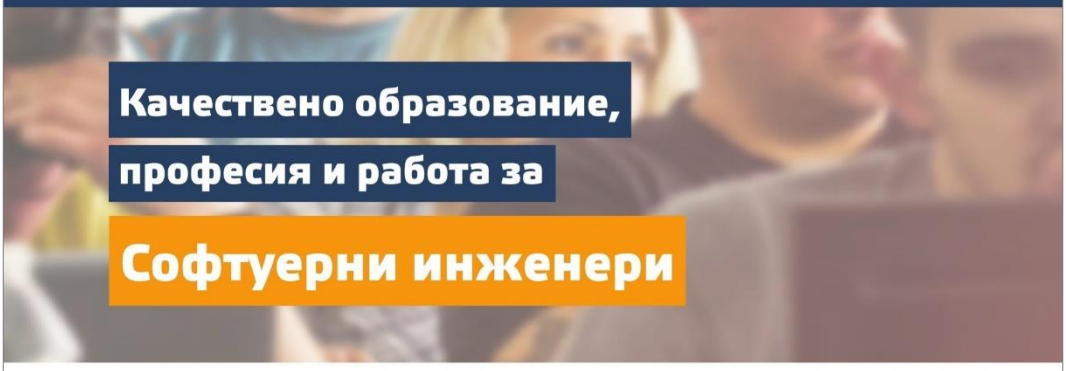
Задачата е да се моделира училище с Java класове. Трябва да декларирате класове заедно с техните полета, свойства, методи и конструктори. Дефинирайте и тестов клас, който демонстрира, че останалите класове работят коректно.

Решения и упътвания

1. Използвайте **enum** за специалностите и университетите.
2. За да избегнете повторение на код извиквайте конструкторите един от друг с **this(<parameters>)**.
3. Използвайте конструктора на класа като място, където броя на обектите от класа **Student** се увеличава.
4. Отпечатайте на конзолата всички полета от класа **Student**, следвани от празен ред.
5. Направете **private** всички членове на класа **Student**, след което използвайки Eclipse (Source -> Generate -> Getters and Setters) дефинирайте автоматично публични методи за достъп до тези полета.
6. Създайте няколко студента и изведете цялата информация за всеки един от тях.
7. Можете да ползвате статичния конструктор, за да създадете инстанциите при първия достъп до класа.
8. Декларирайте три отделни класа: **GSM**, **Battery** и **Display**.
9. Дефинирайте описаните конструктори и за да проверите дали класовете работят правилно направете тестова програма.
10. Направете **private** полето и го инициализирайте в момента на декларацията му.
11. В класовете **GSM**, **Battery** и **Display** дефинирайте подходящи **private** полета и генерирайте getters / setters. Можете да ползвате автоматичното генериране в Eclipse.
12. Добавете метод **printInfo()** в класа **GSM**.
13. Прочетете за класа **ArrayList** в Интернет. Класът **GSM** трябва да пази разговорите си в списък от тип **ArrayList<Call>**.
14. Връщайте като резултат списъка с разговорите.
15. Използвайте вградените методи на класа **ArrayList**.
16. Понеже тарифата е фиксирана, лесно можете да изчислите сумарната цена на проведените разговори.
17. Следвайте директно инструкциите от условието на задачата.

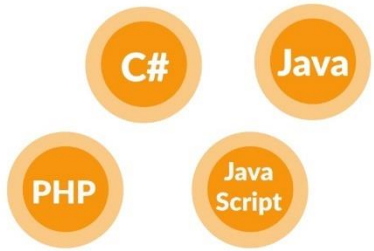
18. Дефинирайте класове **`Book`** и **`Library`**. За списъка с книги ползвайте **`ArrayList<Book>`**.
19. Следвайте директно инструкциите от условието на задачата.
20. Създайте класове **`School`**, **`SchoolClass`**, **`Student`**, **`Teacher`**, **`Discipline`** и в тях дефинирайте съответните им полета, както са описани в условието на задачата. Не ползвайте за име на клас думата "**`Class`**", защото в Java тя има специално значение. Добавете методи за отпечатване на всички полета от всеки от класовете.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 15. Текстови файлове

Автор

Данаил Алексиев

В тази тема...

В настоящата тема ще се запознаем с основните похвати при работа с текстови файлове в Java. Ще разясним какво е това поток, за какво служи и как се ползва. Ще обясним какво е текстов файл и как се чете и пише в текстови файлове. Ще демонстрираме и обясним добрите практики за прихващане и обработка на изключения, възникващи при работата с файлове. Разбира се, всичко това ще онагледим и демонстрираме на практика с много примери.

Потоци

Потоците (streams) са важна част от всяка входно-изходна библиотека. Те намират своето приложение, когато програмата трябва да "прочете" или "запише" данни от или във външен източник на данни – файл, други компютри, сървъри и т.н.

Преди да продължим е важно да уточним, че терминът **вход (input)** се асоциира с четенето на информация, а терминът **изход (output)** – със записването на информация.

Какво представляват потоците?

Потокът е наредена последователност от байтове, които се изпращат от едно приложение или входно устройство и се получават в друго приложение или изходно устройство. Тези байтове се изпращат и получават един след друг и винаги пристигат в същия ред, в който са били изпратени. Потоците са абстракция на комуникационен канал за данни, който свързва две устройства или програми.

Потоците са основното средство за обмяна на информация в компютърния свят. Чрез тях различни програми достъпват файловете на компютъра, чрез тях се осъществява и мрежова комуникация между отдалечени компютри. За да прочетем или запишем нещо от или във файл, единственото, което трябва да направим, е да отворим поток към дадения файл. Това ни позволява да достъпим данните, записани във файла, и да извършим необходимата ни операция (вход или изход).

Модерните сайтове в Интернет не могат без потоци и така наречения **streaming** (произлиза от stream - поток), който представлява достъпване на големите мултимедийни файлове по уеб страниците чрез поток. Това позволява техния преглед да започне преди цялостното им сваляне, което повишава бързодействието на страницата.

Основни неща, които трябва да знаем за потоците

Потоците се използват, за да четем и записваме данни от и на различни устройства. Те улесняват комуникацията между програма и файл, програма и отдалечен компютър и т.н.

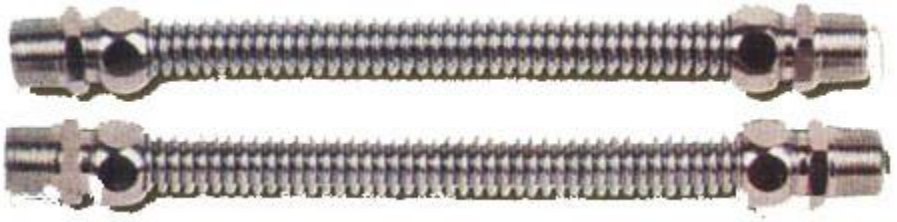
Потоците са **подредени** серии от байтове. Не случайно наблягаме на думата подредени. От огромна важност е да се запомни, че потоците са строго подредени и организирани. По никакъв начин не можем да си позволим да влияем на подредбата на информацията в потока, защото по този начин ще я направим неизползваема.

Потоците позволяват **последователен** достъп до данните си. Отново е важно да се вникне в значението на думата последователен. Може да манипулираме данните само в реда, в който те пристигат от потока. Това е тясно свързано с горното свойство. Имайте това предвид, когато създавате собствени програми. Не можете да вземете първия байт, след това осмия, третия, тринадесетия и така нататък. Потоците **не** предоставят произволен достъп до данните си, а само **последователен**. Ако ви се струва по-лесно, може да мислим за потоците като за свързан списък от байтове, в който те имат строга последователност.

За различните ситуации има различни видове потоци. Едни служат за работа с текстови файлове, други – за работа с бинарни (двоични) файлове, трети пък – за работа със символни низове. Различни са и потоците, които се използват при мрежова комуникация. Голямото изобилие от потоци ни улеснява в различните ситуации, но също така и ни затруднява, защото трябва да сме запознати със спецификата на всеки отделен тип, преди да го използваме в приложението си.

Потоците се отварят преди началото на работата с тях и се затварят след като е приключило използването им. Това е нещо абсолютно задължително и не може да се пропусне, поради риск от загуба на данни, повреждане на файла, към който е отворен потока и т.н. – все неприятни неща, които не трябва да допускаме да стават в програмите ни.

Потоците можем да оприличим на тръби, свързващи две точки:



От едната страна "наливаме" данни, а от другата данните "изтичат". Този, който налива данните не се интересува как те се пренасят, но е сигурен, че каквото е налял, такова ще излезе от другата страна на тръбата. Тези, които ползват потоците не се интересуват как данните стигат до тях. Те знаят, че ако някой нещо налее от другата страна, то ще пристигне при тях. Потоците са транспортен канал за данни, както и тръбите.

Потоци в Java – основни класове

В Java класовете за работа с потоци се намират в пакета **java.io**. Сега ще се концентрираме върху тяхната йерархия и организация.

Можем да отличим два основни типа потоци – такива, които работят с двоични данни и такива, които работят с текстови данни. Ще се спрем на основните характеристики на тези два вида след малко.

Общото между тях е организацията и структурирането им. На върха на йерархията стоят абстрактни класове съответно за вход и изход. Те няма как да бъдат инстанцирани, но дефинират основната функционалност, която притежават всички останали потоци. Съществуват и буферирани потоци, които не добавят никаква допълнителна функционалност, но позволяват работата с буфер при четене и записване на информацията, което значително повишава бързодействието. Буферираните потоци няма да се разглеждат в тази глава, тъй като ние се концентрираме върху обработката на текстови файлове. Ако имате желание, може да се допитате до богатата документация, достъпна в Интернет, или към някой учебник за по-напреднали в програмирането.

Основните класове в пакета **java.io** са **InputStream**, **OutputStream**, **BufferedInputStream**, **BufferedOutputStream**, **DataInputStream**, **DataOutputStream**, **Reader**, **Writer**, **BufferedReader**, **BufferedWriter**, **PrintWriter** и **PrintStream**. Сега ще се спрем по-обстойно на тях, разделяйки ги по основния им признак – типа данни, с които работят.

В тази глава в примерите за писане в текстов файл ще ползваме само **PrintStream**, защото е идеален за работа с текстови файлове и с него се борави изключително лесно.

Всички потоци в Java си приличат и по едно основно нещо – задължително е да ги затворим, след като сме приключили работа с тях. В противен случай рискуваме да навредим на данните в потока или файла, към който

сме го отворили. Това ни води и до първото основно правило, което винаги трябва да помним при работа с потоци:

Винаги затваряйте потоците и файловете, с които работите! Оставянето на отворен поток или файл води до загуба на ресурси и може да блокира работата на други потребители или процеси във вашата система.

Двоични и текстови потоци

Както споменахме по-рано, можем да разделим потоците на две големи групи в съответствие с типа данни, с който боравят, а именно – двоични потоци и текстови потоци.

Двоични потоци

От името им личи, че работят с двоични данни. Сами се досещате, че това ги прави универсални и тях може да ползваме за четене на информация от всякакви файлове (картинки, музикални и мултимедийни файлове, текстови файлове и т.н.). Ще ги разгледаме накратко, защото за момента се ограничаваме до работа с текстови файлове.

Двата основни абстрактни класа, които седят в дъното на йерархията на двоичните потоци са съответно **`InputStream`** за вход и **`OutputStream`** за изход. Както казахме по-горе, те не могат да се инстанцират, но дефинират основната функционалност и се ползват при създаването на другите двоични потоци.

`InputStream` ни предлага различни методи за четене (четене само на един байт, четене на произволен брой байтове и записването им в масив и т.н.), пропускане на определен брой байтове, проверяване на броя достъпни байтове и, разбира се, метод за затваряне на потока. Обект от този клас може да получим от източник, извиквай съответния му метод.

`OutputStream` също е абстрактен клас. Той съдържа базовата функционалност за записване на информация в даден източник. Основните му методи ни позволяват да записваме различен брой байтове в източника, да прочистваме буфера на потока и, отново, да затваряме потока.

Другите класове за работа с бинарни потоци са **`BufferedInputStream`**, **`BufferedOutputStream`**, **`DataInputStream`**, **`DataOutputStream`**. За да можем да създадем обекти от тях, се нуждаем от обект от **`InputStream`** или, съответно, **`OutputStream`**.

Текстови потоци

Текстовите потоци са много подобни на двоичните, но работят само с текстови данни или, по-точно казано, с данни от символен тип (**`char`**). Идеални са за работа с текстови файлове. От друга страна това ги прави неизползваеми при работа с каквито и да е бинарни файлове.

Основните класове за работа с текстови потоци са **Reader** и **Writer**. Те са аналогични на основните класове от двоичните потоци. Методите им са същите, но вместо аргументи от тип **byte** приемат **char**. Както знаете, символите в Java са Unicode символи, но потоците могат да работят освен с Unicode и с други кодирания.

Съществуват и буферираните варианти **BufferedReader** и **BufferedWriter**, които се отличават с по-голямото си бързодействие.

Важно място има и класа **PrintWriter**, но той не е във фокуса на тази глава. Ако имате желание, може да погледнете документацията на Java API-то или източници в Интернет.

Класът, на когото ще обърнем най-голямо внимание е **PrintStream**. Той в голяма част се припокрива с **PrintWriter**, но има някой специфични особености. За да създадем обект от **PrintStream** класа ни е нужен файл или символен низ с име и път до файла. Той има много полезни методи, като например добре познатите **print(…)** и **println(…)**. Всъщност **System.out** не е нищо повече от обект от тип **PrintStream**. Ето защо боравейки с този клас ще можем да използваме всички методи, с които вече сме добре запознати от работата ни с конзолата. Това, с което **PrintStream** класа се отличава, е, че скрито от нас той превръща текста в байтове преди да ги запише на желаното място. Леснотата, с която се работи с него и големите му възможности го правят идеален за използване в примерите, които ще последват по-напред в тази глава.

Четене от текстов файл

Текстовите файлове предоставят идеалното решение за записване на данни, които трябва да ползваме често, а са твърде обемисти, за да ги въвеждаме ръчно всеки път, когато стартираме програмата. Сами се убеждавате, че тази практика изисква идеално владение на механизмите за четене на текстови файлове.

Java платформата предоставя множество начини за четене от файлове, но не всички са много лесни и интуитивни за използване. Ето защо се спираме на нещо познато за вас – класът **java.util.Scanner**. Сигурно до сега стотици пъти ви се е налагало да го ползвате за всевъзможни операции. Именно за това считаме, че ще е идеален за случая, защото е най-лесния начин за четене на текстов файл и същевременно сте имали много шансове да го усвоите до съвършенство.

Класът java.util.Scanner за четене на текстов файл

В момента сигурно сте малко объркани. До тук казахме, че четенето и записването в текстови файлове става само и изключително с потоци, а, същевременно, **java.util.Scanner** не се появи никъде в изброените по-горе потоци и не сте сигурни дали въобще е поток. Наистина, **java.util.Scanner** не е поток, но може да работи с потоци. Той предоставя най-лесния и

разбираем начин за четене от текстов файл като се има предвид, че често до сега сте го използвали за четене на различни неща от конзолата.

Едно от големите предимства на **java.util.Scanner** е, че не е нужно да има поток, за да бъде създаден. Може да го създадем просто от файл, което значително ни улеснява и намалява вариантите за грешка. При създаването можем да уточним и кодирането. Ето пример как може да бъде създаден обект от класа **java.util.Scanner**:

```
// Link the File variable to a file on the computer
File file = new File("test.txt");

// Create a Scanner connected to a file and specify encoding
Scanner fileReader = new Scanner(file, "windows-1251");

// Read file here...

// Close the resource after you've finished using it
fileReader.close();
```

Първото, което трябва да направим е да създадем променлива от тип **java.io.File**, която да свържем с конкретен файл от нашия компютър. За целта е нужно само да подадем като параметър в конструктора му името на желания файл. Имайте предвид, че ако файлът се намира в папката на проекта, то можем да подадем само конкретното му име. В противен случай трябва да подадем пълния път до файла.

Не забравяйте при подаване на пълния път до даден файл да направите escaping на наклонените черти, които се използват за разделяне на папките (`"C:\\Temp\\test.txt"`, а не `"C:\Temp\test.txt"`). По възможност избягвайте пълни пътища и работете с релативни!

Използването на пълен път до даден файл (примерно **`C:\Temp\test.txt`**) е лоша практика, защото прави програмата ви зависима от средата и непреносима. Ако я пренесете на друг компютър, ще трябва да коригирате пътищата до файловете, които тя търси. Ако използвате релативен (относителен) път спрямо текущата директория (това е директорията на проекта), вашата програма ще е лесно преносима.

Вече можем да създадем и нашия **java.util.Scanner**. Като параметри този път подаваме новосъздадената файлова променлива и име на encoding (като **String**), който желаем да ползваме при прочитането на файла (в този случай използваме **windows-1251**). Така можем да го ползваме за прочитане на желаната информация. Ако не укажем изрично кодиране, Java използва кодирането по подразбиране в операционната система (което може да е различно на различни компютри).

Ще забележите, че при създаването на нашия **Scanner**, че Eclipse ви предупреждава за неприхванатата изключителна ситуация. За сега изберете опцията просто да добавите **throws** декларация. За прихващането и обработването на изключителни ситуации при работа с файлове ще стане дума малко по-късно в тази глава, в секцията "Обработка на грешки".

Четене на текстов файл ред по ред – пример

След като се научихме как да създаваме **Scanner** вече можем да се опитаме да направим нещо по-сложно: да прочетем цял текстов файл ред по ред и да печатаме прочетеното на конзолата. Моят съвет е да използваме възможността на Eclipse за създаване на текстови файлове (десен бутон на мишката върху проекта -> New -> File; за име пишем нещо, което завършва на **.txt**), за да създадем нашия текстов файл. Така той ще се създаде в самия проект и няма да се налага да подаваме пълния път до него при създаването на файлова променлива. Нека нашият файл изглежда така:

sample.txt

```
This is our first line.
This is our second line.
This is our third line.
This is our fourth line.
This is our fifth line.
```

Имаме текстов файл, от който да четем. Сега трябва да създадем файлова променлива, свързана с него, да създадем и **Scanner** и да прочетем и отпечатаме всички редове. Това можем да направим по следния начин:

FileReader.java

```
// Link the File variable to a file on the computer
File file = new File("sample.txt");

// Next line may throw an exception!
Scanner fileReader = new Scanner(file);

int lineNumber = 0;

// Read file
while (fileReader.hasNextLine()) {
   lineNumber++;
   System.out.printf("Line %d: %s%n",
      lineNumber, fileReader.nextLine());
}

// Close the resource after you've finished using it
```

```
fileReader.close();
```

Сами се убеждавате, че няма нищо трудно в четенето на текстови файлове. Първата част на програмата вече ни е добре позната – създаваме файловата променлива, а след това и **Scanner**. След това създаваме и една променлива – брояч, чиято цел е да брои и показва на кой ред от файла се намираме в текущия момент.

За същинската част – прочитането на файла ред по ред, **while** цикъл. За условие за изпълнение на цикъла използваме метода на класа **Scanner** – **hasNextLine()**. Той проверява дали има следващ достъпен ред или е достигнат края на файла и връща резултата от тип **boolean**. Съществуват подобни методи за много от Java типовете. В тялото на цикъла задачата ни се свежда до това да увеличим стойността на променливата – брояч с единица и след това да отпечатаме текущия ред от файла в желания от нас формат. За целта използваме един метод, който ни е отлично познат от задачите, в които се е изисквало да се прочете нещо от конзолата – **nextLine()**.

След като сме прочели нужното ни от файла, отново не бива да забравяме да затворим **Scanner** обекта, за да избегнем загубата на ресурси. За това ползваме метода **close()**.

Винаги затваряйте инстанциите на `Scanner` след като приключите работа с тях. В противен случай рискувате да загубите системни ресурси. За затваряне използвайте метода `close()`.

Резултатът от изпълнението на програмата би трябвало да изглежда така:

```
Line 1: This is our first line.
Line 2: This is our second line.
Line 3: This is our third line.
Line 4: This is our fourth line.
Line 5: This is our fifth line.
```

Кодиране на файловете. Четене на кирилица

Нека сега разгледаме проблемите, които се появяват при четене с некоректно кодиране, например при четене на файл на кирилица.

Кодиране (encoding)

Добре знаете, че в паметта на компютрите всичко се запазва в двоичен вид. Фактически, това означава, че се налага и текстовите файлове да се представят цифрово, за да могат да бъдат съхранени в паметта. Този процес се нарича **кодиране на файловете**.

Кодирането се състои в заместването на текстовите символи (цифри, букви, препинателни знаци и т.н.) с точно определени числови стойности. Може грубо да си го представите като голяма таблица, в която срещу всеки символ стои определена стойност (пореден номер).

Пример за кодираща схема (encoding или charset) е например **`ISO 8859-1`**, **`windows-1251`**, **`UTF-8`**, **`KOI8-R`** и т.н. Това е една таблица със символи и техните номера, но може да съдържа и специални правила. Например символът "ударение" (U+0300) е специален и се залепя за последния символ, който го предхожда.

Четене на кирилица

Вероятно вече се досещате, че ако искаме да четем от файл, който съдържа символи от кирилицата, трябва да използваме точния encoding, който "разбира" тези специални символи. Такъв именно е **`windows-1251`**. С него спокойно можем да четем текстови файлове, съдържащи кирилица. Единственото, което трябва да направим, е да го определим като encoding на потока, който ще обработваме с нашия **`Scanner`** (погледнете отново примера за създаване на **`Scanner`**).

Ако не укажем изрично кодиращата схема (encoding) за четене от файла, ще бъде използван системният encoding, който е различен на всеки един компютър. В такъв случай програмата може и да работи коректно, но може и да не работи коректно. Може да се случи и нещо по-лошо: при нас програмата да работи коректно, а като я занесем при клиента, за който е предназначена, там да се счупи кирилицата.

Може би се чудите какво става, ако "омажем" кодирането при четене или писане във файл. Възможни са няколко сценария:

- Ако ползваме само латиница, всичко ще работи нормално.
- Ако ползваме кирилица и четем с грешен encoding, ще прочетем т. нар. каракацили (познати още като джуджуфлечки или маймуняци). Това са случайни безсмислени символи.
- Ако записваме кирилица в кодиране, което не поддържа кирилската азбука, буквите от кирилицата ще бъдат заменени безвъзвратно със символа "**`?`**" (въпросителна).

При всички случаи това са неприятни проблеми, които можем да не забележим веднага, а чак след време.

За да избегнете проблемите с неправилно кодирането на файловете винаги задавайте encoding изрично. Иначе програмата може да работи некоректно или да се счупи след време, без въобще да я бутате.

Стандартът Unicode. Четене на Unicode

Unicode представлява индустриален стандарт, който позволява на компютри и други електронни устройства винаги да представят и манипулират по един и същи начин текст, написан на повечето от световните писмености. Той се състои от дефиниции на над 100 000 символа, както и разнообразни стандартни кодиращи схеми (encodings). Обединението на различните символи, което ни предлага Unicode, води до голямото му разпространение. Както знаете, символите в Java (типовете **char** и **String**) също се представят в Unicode.

За да прочетем символи, записани в Unicode, трябва да използваме някоя от поддържаните в този стандарт кодиращи схеми (encodings). Най-известен и широко използван е UTF-8. Той представя стандартните ASCII символи с 1 байт, а всички останали – с до 4 байта. Можем да го определим за encoding по вече познатия ни начин (погледнете отново примера за създаване на **Scanner**):

```
File file = new File("sample.txt");
Scanner scanner = new Scanner(file, "UTF-8");
```

Ако се чудите дали за четене на текстов файл на кирилица да ползвате кодиране **windows-1251** или **UTF-8**, на този отговор няма ясен отговор. И двата стандарта масово се ползват за записване на текстове на български език. И двете кодиращи схеми за позволени и може да ги срещнете.

Писане в текстов файл

Писането в текстови файлове е много удобен способ за съхранение на различни видове информация. Например, можем да записваме резултатите от изпълнението на дадена програма. Също така можем да ги ползваме, примерно и за да направим нещо като дневник на програмата – удобен начин за следене кога се е стартирала, отбелязване на различни грешки при изпълнението и т.н.

Отново както и при четенето текстов файл, и при писането ще използваме един познат ни от работата с конзолата клас, въпреки че този път това не е толкова явно. Вярвам, че сте добре запознати с **System.out**. Това не е нищо повече от инстанция на класа, който ще използваме за писане в текстови файлове, а именно **java.io.PrintStream**.

Класът java.io.PrintStream

Както вече няколкократно споменахме, класът **PrintStream** е част от пакета **java.io** и се използва изключително и само за работа с текстови данни. За разлика от другите текстови потоци, преди да запише данните на желаното място, той ги превръща в байтове. **PrintStream** ни дава възможност при

създаването си да определим желания от нас encoding. Можем да създадем инстанция на класа по следния начин:

```
PrintStream fileWriter = new PrintStream(
   "test.txt", "windows-1251");
```

Като параметри на конструктора трябва да подадем файл/име на файл и ако искаме, желаният от нас encoding. Този ред код отново може да предизвикат появата на грешка. За сега просто добавете **throws** декларация в сигнатурата на текущия метод. Скоро ще обърнем внимание и на обработката на грешки при работа във файлове.

Отпечатване на числата от 1 до 20 в текстов файл – пример

След като вече можем да създаваме **PrintStream**, ще го използваме по предназначение. Целта ни ще е да запишем в един текстов файл числата от 1 до 20, като всяко число е на отделен ред. Можем да го направим по следния начин:

```
// Create a PrintStream instance
PrintStream fileWriter = new PrintStream("numbers.txt");

// Loop through the numbers from 1 to 20 and write them
for (int num = 1; num <= 20; num++) {
   fileWriter.println(num);
}

// Close the stream when you are done using it
fileWriter.close();
```

Започваме като създаваме инстанция на **PrintStream** по вече познатия ни от примера начин.

За да вземем числата от 1 до 20 ще използваме един **for**-цикъл. В тялото на цикъла ще използваме метода **println(…)**, който отново познаваме от работата ни с конзолата, за да запишем текущото число на нов ред във файла. Не бива да се притеснявате, ако файлът, чието име сте дали не съществува. Ако случаят е такъв, той ще бъде автоматично създаден в папката на проекта, а ако вече съществува, ще бъде презаписан (ще изгубите старото му съдържание). Той ще има следния вид:

numbers.txt

```
1
2
3
```

```
…
20
```

В края на програмата затваряме потоците, които сме използвали.

Не пропускайте да затворите потока след като приключите с използването му! За затваряне използвайте метода `close()`.

Когато искате да печатате кирилица и се чудите кое кодиране да ползвате, предпочитайте кодирането UTF-8. То е универсално и поддържа не само кирилица, но и всички широкоразпространени световни азбуки: гръцки, арабски, китайски, японски и др.

Обработка на грешки

Докато сте правили примерите до тук, сигурно сте забелязали, че при доста от операциите, свързани с файлове, могат да възникнат изключителни ситуации. Основните принципи и подходи за тяхното прихващане и обработка вече са ви познати от предишните глави и най-вече от главата "Обработка на изключения". Сега ще се спрем малко на специфичните грешки при работа с файлове и най-добрите практики, за тяхната обработка.

Прихващане на изключения при работа с файлове

Може би най-често срещаната грешка при работа с файлове е **`FileNotFoundException`** (от името и личи, че индикира, че желаният файл не е намерен). Тя може да възникне, когато използваме този файл за създаването на **`Scanner`** или **`PrintStream`**.

Когато избираме и типа encoding при създаване на **`Scanner`** и **`PrintStream`** може да възникне и **`UnsupportedEncodingException`**. Това значи, че избраният от нас encoding не е поддържан.

Друга често срещана грешка е **`IOException`**. Това е клас, който е базов за всички входно-изходни грешки при работа с потоци.

Стандартният подход при обработване на изключения при работа с файлове е следният: декларираме променливите от клас **`Scanner`** и/или **`PrintStream`** преди **`try-catch-finally`** блока, като ги инициализираме със стойност **`null`**. В самия блок ги инициализираме с нужните ни стойности и прихващаме и обработваме потенциалните грешки по подходящ начин. За по-специална цел ще ползваме **`finally`** частта. За да онагледим казаното до тук, ще дадем пример.

Прихващане на грешка при отваряне на файл – пример

Ето как можем да прихванем изключенията, настъпващи при работа с файлове:

```
String fileName = "sample.txt";
Scanner fileReader = null;
int lineNumber = 0;
try {
   fileReader = new Scanner(new File(fileName));
   System.out.println("File " + fileName + " opened.");

   while (fileReader.hasNextLine()) {
      lineNumber++;
      System.out.printf("Line %d:%s%n",
         lineNumber, fileReader.nextLine());
   }
} catch (FileNotFoundException fnf) {
   System.out.println("File " + fileName + " not found.");
} catch (NullPointerException npe) {
   System.out.println("File " + fileName + " not found.");
} finally {
   // Close the scanner in the finally block
   if (fileReader != null) {
      fileReader.close();
   }
   System.out.println("Scanner closed.");
}
```

Това е същият пример за четене на файл ред по ред. Този път сме прихванали и обработили по подходящ начин възможните извънредни ситуации. Обърнете внимание как използваме **`finally`** блока. Тъй като той се изпълнява винаги, независимо дали са възникнали грешки или не, той е идеалното място където да направим задължителното затваряне на използваните ресурси (в случай, че не са вече затворени). За подсигуряване сме добавили и **`catch`** блок за **`NullPointerException`**.

Текстови файлове – още примери

Надяваме се теоретичните обяснения и примерите до сега да са успели да ви помогнат да навлезете в тънкостите при работа с текстови файлове. Сега ще разгледаме още няколко по-комплексни примери с цел да затвърдим получените до тук знания и да онагледим как да ги ползваме в практически задачи.

Брой срещания на дума във файл – пример

Ето как може да реализираме проста програма, която брои колко пъти се среща дума в даден текстов файл (за дума считаме всеки подниз от текста). В примера, нека текстът изглежда така:

sample.txt

```
This is our "Intro to Programming in Java" book.
In it you will learn the basics of Java programming.
You will find out how nice Java is.
```

Броенето можем да направим така: ще прочитаме файла ред по ред и всеки път, когато срещнем търсената от нас дума, ще увеличаваме стойността на една променлива (брояч). Ще обработим възможните изключителни ситуации, за да може потребителят да получава адекватна информация при появата на грешки. Ето и примерна реализация:

CountWordOccurrences.java

```
String fileName = "sample.txt";
Scanner fileReader = null;
int occurrences = 0;
String word = "Java";
try {
   fileReader = new Scanner(new File(fileName));
   System.out.println("File " + fileName + " opened.");

   while (fileReader.hasNextLine()) {
      String line = fileReader.nextLine();
      int index = line.indexOf(word);
      while (index != -1) {
         occurrences++;
         index = line.indexOf(word, (index + 1));
      }
   }
} catch (FileNotFoundException fnf) {
   System.out.println("File " + fileName + " not found.");
} catch (NullPointerException npe) {
   System.out.println("File " + fileName + " not found.");
} finally {
   if (fileReader != null) {
      fileReader.close();
   }
   System.out.println("Scanner closed.");
}

System.out.printf("The word %s occurs %d times",
```

```
   word, occurrences);
```

За леснота в примера думата, която търсим е твърдо кодирана (hardcoded). Вие може да реализирате програмата така, че да търси дума, въведена от потребителя.

Виждате, че примерът не се различава много от предишните. Отново инициализираме променливите извън **try-catch-finally** блока. Пак използваме **while**-цикъл, за да прочитаме редовете на текстовия файл един по един. Вътре в тялото му има още един **while**-цикъл, с който преброяваме колко пъти се среща думата в дадения ред и увеличаваме брояча на срещанията. Това става като използваме метода от класа **String indexOf(…)** (припомнете си какво прави той в случай, че сте забравили). Не пропускаме да затворим **Scanner** обекта. Единственото, което после ни остава да направим е да изведем резултата на конзолата.

За нашия пример резултатът е следният:

```
File sample.txt opened.
Scanner closed.
The word Java occurs 3 times
```

Коригиране на файл със субтитри – пример

Сега ще разгледаме един по-сложен пример, в който едновременно четем от един файл и записваме в друг. Става дума за програма, която коригира файл със субтитри за някакъв филм.

Нашата цел ще бъде да изчетем един файл със субтитри, които са некоректни и не се появяват в точния момент и да отместим времената по подходящ начин, за да се появяват правилно. Един такъв файл в общия случай съдържа времето за появяване на екрана, времето за скриване от екрана и текста, който трябва да се появи в дефинирания интервал от време. Ето как изглежда един типичен файл със субтитри:

GORA.sub

```
{1029}{1122}{Y:i}Капитане, системите са|във шибана готовност.
{1123}{1270}{Y:i}Шибаното налягане е стабилно.|- Пригответе се за
шибаното кацане.
{1343}{1468}{Y:i}Моля, затегнете коланите|и се настанете по местата
си.
{1509}{1610}{Y:i}Координати 5.6|- Пет, пет, шест, точка ком.
{1632}{1718}{Y:i}Къде се дянаха|шибаните координати?
{1756}{1820}Командир Логар,|всички говорят на английски.
{1821}{1938}Не може ли да преминем|на турски още от началото?
{1942}{1992}Може!
```

```
{3104}{3228}{Y:b}Г.О.Р.А.|филм за космоса
...
```

За да го коригираме, просто трябва да нанесем корекция във времето за показване на субтитрите. Такава корекция може да бъде отместване (добавяне или изваждане на някаква константа) или промяна на скоростта (умножаване по някакви число, примерно 1.05).

Ето и примерен код, с който може да реализираме такава програма:

FixingSubtitles.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.io.UnsupportedEncodingException;
import java.util.Scanner;

public class FixingSubtitles {
   private static final int COEFFICIENT = 2;
   private static final int ADDITION = 5000;
   private static final String INPUT_FILE = "GORA.sub";
   private static final String OUTPUT_FILE = "fixed.sub";

   public static void main(String[] args) {
      Scanner fileInput = null;
      PrintStream fileOutput = null;
      try {
         // Create scanner with the Cyrillic encoding
         fileInput = new Scanner(
            new File(INPUT_FILE), "windows-1251");
         // Create PrintWriter with the Cyrillic encoding
         fileOutput = new PrintStream(
            OUTPUT_FILE, "windows-1251");
         String line;
         while (fileInput.hasNextLine()) {
            line = fileInput.nextLine();
            String fixedLine = fixLine(line);
            fileOutput.println(fixedLine);
         }
      } catch (FileNotFoundException fnfe) {
         System.err.println(fnfe.getMessage());
      } catch (UnsupportedEncodingException uee) {
         System.err.println(uee.getMessage());
      } finally {
         if (null != fileInput) {
            fileInput.close();
         }
```

```
            if (null != fileOutput) {
               fileOutput.close();
            }
         }
      }

      private static String fixLine(String line) {
         // Find closing brace
         int bracketFromIndex = line.indexOf('}');

         // Extract 'from' time
         String fromTime = line.substring(1, bracketFromIndex);

         // Calculate new 'from' time
         int newFromTime =
            Integer.parseInt(fromTime) * COEFFICIENT + ADDITION;

         // Find the following closing brace
         int bracketToIndex = line.indexOf('}', bracketFromIndex+1);

         // Extract 'to' time
         String toTime =
            line.substring(bracketFromIndex + 2, bracketToIndex);

         // Calculate new 'to' time
         int newToTime =
            Integer.parseInt(toTime) * COEFFICIENT + ADDITION;

         // Create a new line using the new 'from' and 'to' times
         String fixedLine = "{" + newFromTime + "}" + "{" +
            newToTime + "}" + line.substring(bracketToIndex + 1);

         return fixedLine;
      }
   }
```

Тук създаваме **Scanner** и **PrintStream** и задаваме да използват encoding "**windows-1251**", защото ще работим с файлове, съдържащи кирилица. Това значи, че не трябва да забравяме да добавим **catch** блок, за да прихванем **UnsupportedEncodingException**. Отново използваме вече познатия ни начин за четене на файл ред по ред. Различното този път е, че в тялото на цикъла записваме всеки ред във файла с вече коригирани субтитри, след като го поправим в метода **fixLine(String)** (този метод не е обект на нашата дискусия, тъй като може да бъде имплементиран по много и различни начини в зависимост какво точно искаме да коригираме). Важно е да на забравим да затворим потоците във **finally** блока.

Упражнения

1. Напишете програма, която чете от текстов файл и принтира нечетните му редове на конзолата.
2. Напишете програма, която чете списък от имена от един текстов файл, сортира ги по азбучен ред и ги запазва в друг файл. Имената да са с латински букви. На всеки ред от файла, където са записани имената, има точно по едно име. На всеки ред от файла с резултата също трябва да има само по едно име.
3. Напишете програма, която чете от файл квадратна матрица от цели числа и намира подматрицата с размери 2 x 2 с най-голяма сума и записва тази сума в отделен текстов файл. Първия ред на входния файл съдържа големината на записаната матрица (N). Следващите N реда съдържат по N числа, разделени с интервал.
4. Напишете програма, която заменя всяко срещане на подниза "start" с "finish" в текстов файл. Можете ли да пренапишете програмата така, че да заменя само цели думи?
5. Напишете програма, която прочита списък от думи от файл, наречен **`words.txt`**, преброява колко пъти всяка от тези думи се среща в друг файл **`text.txt`** и записва резултата в трети файл – **`result.txt`**, като преди това ги сортира по брой на срещане в намаляващ ред.

Решения и упътвания

1. Използвайте примерите, които разгледахме в настоящата глава.
2. Записвайте всяко прочетено име в масив и след това го сортирайте по подходящ начин.
3. Прочетете първия ред от файла и създайте матрица с получения размер. След това четете останалите редове един по един и отделяйте числата. След това ги записвайте на съответния ред в матрицата.
4. Четете файла ред по ред и използвайте методите на класа **`String`**.
5. Създайте хеш таблица с ключ думите от **`words.txt`** и стойност броя срещания на всяка дума. Четете ред по ред **`text.txt`** и разделяйте всеки ред на думи. Проверете дали някоя от получените при разделянето думи се среща в хеш таблицата и при нужда прибавяте 1 към броя на срещанията ѝ.

Глава 16. Линейни структури от данни

Автори

Цвятко Конов

Светлин Наков

В тази тема...

Много често, когато решаваме някоя задача, ни се налага да използваме определена съвкупност от данни (например масив). В зависимост от конкретната задача ни се налага да прилагаме различни операции върху тази съвкупност от данни. В настоящата тема ще се запознаем с някои от основните представяния на данните в програмирането. Ще видим как при определена задача една структура е по-ефективна и удобна от друга. Ще разгледаме структурите "списък", "стек" и "опашка" и тяхното приложение. Подробно ще се запознаем и с някои от реализациите на тези структури.

Абстрактни структури от данни

Преди да се запознаем с класовете в Java, които ни позволяват по лесен начин да работим с някои често използвани структури от данни (като списъци и хеш-таблици), нека първо разгледаме понятията "структури от данни" и "абстрактни структури от данни".

Какво е структура данни?

Много често, когато пишем програми ни се налага да работим с множество от обекти (данни). Понякога добавяме и премахваме елементи, друг път искаме да ги подредим или да обработваме данните по друг специфичен начин. Поради това са изработени различни начини за съхранение на данните в зависимост от задачата, като най-често между елементите съществува някаква наредба (например обект А е преди обект Б).

В този момент на помощ ни идват **структурите данни** – множество от данни организирани на основата на логически и математически закони. Много често избора на правилната структура прави програмата много по-ефективна – можем да спестим памет и време за изпълнение.

Какво е абстрактен тип данни?

Най-общо **абстрактният тип данни (АТД)** дава определена дефиниция (абстракция) на конкретната структура т.е. определя допустимите операции и свойства, без да се интересува от конкретната реализация. Това позволява един тип абстрактни данни да има различни реализации, респективно различна ефективност.

Основни структури от данни в програмирането

Могат ясно да се различат няколко групи структури:

- Линейни – към тях спадат списъците, стековете и опашките
- Дървовидни – различни типове дървета
- Речници – хеш-таблици
- Множества

В настоящата тема ще разгледаме линейните (списъчни) структури от данни, а в следващите няколко теми ще обърнем внимание и на по-сложните структури като дървета, графи, хеш-таблици и множества и ще обясним кога се използва и прилага всяка от тези структури.

Овладяването на основните структури данни в програмирането е от изключителна важност, тъй като без тях не можете да програмирате ефективно. В основата на програмирането стоят структурите от данни и алгоритмите, с които малко по малко ви запознаваме в настоящата книга.

Списъчни структури

Най–често срещаните и използвани са линейните (списъчни) структури. Те представляват абстракция на всякакви видове редици, последователности, поредици и други подобни от реалния свят.

Списък

Най–просто можем да си представим списъка като редица от елементи. Например покупките от магазина или задачите за деня представляват списъци. В списъка можем да четем всеки един елементите напр. покупките, както и да добавяме нови покупки в него. Можем спокойно да задраскваме (изтрием) покупки или да ги разместваме.

Абстрактна структура данни "списък"

Нека сега дадем една по-строга дефиниция на структурата списък:

Списък е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елементите му са наредени последователно.

Списъкът позволява добавяне елементи на всяко едно място както и премахването им, както и последователното им обхождането. Както споменахме по-горе един АТД може да има няколко реализации. Пример за такъв АТД е интерфейсът **java.util.List**.

Интерфейсите в Java изграждат една "рамка" за техните имплементации – класовете. Тази рамка представлява съвкупност от методи и свойства, които всеки клас, имплементиращ интерфейса, трябва да реализира (типът данни "интерфейс" в Java ще дискутираме подробно в главата "Принципи на обектно-ориентираното програмиране").

Всеки АТД реално определя някакъв интерфейс. Нека разгледаме интерфейса **java.util.List**. Основните методи, които той декларира, са:

- **void add(int, Object)** - добавя елемент на предварително избрана позиция
- **boolean contains(Object)** – проверява дали елемента се съдържа в списъка
- **Object get(int)** – взима елемента на съответната позиция
- **boolean isEmpty()** – проверява дали списъка е празен
- **boolean remove(Object)** – премахва съответния елемент
- **Object remove(int)** – премахва елемента на дадена позиция
- **int indexOf(Object)** – връща позицията на елемента

Нека видим няколко от основните реализации на АТД списък и обясним в какви ситуации се използва всяка от тях.

Статичен списък (реализация чрез масив)

Масивите изпълняват много от условията АТД списък, но имат една съществена разлика – списъците позволяват добавяне на нови елементи, докато масивите имат фиксиран размер.

Въпреки това е възможна реализация на списък чрез масив, който автоматично увеличава размера си при нужда (по подобие на класа **StringBuilder**). Такъв списък се нарича **статичен**. Ето една имплементация на статичен списък, реализиран чрез разширяем масив:

```
public class CustomArrayList {
   private Object[] arr;
   private int count;
   private static final int INITIAL_CAPACITY = 4;

   /** Initializes the array-based list – allocate memory **/
   public CustomArrayList() {
      arr = new Object[INITIAL_CAPACITY];
```

```
      count = 0;
   }

   /**
    * @return the actual list length
    */
   public int getLength() {
      return count;
   }
}
```

Първо си създаваме масива, в който ще пазим елементите, както и брояч за това колко имаме в момента. След това добавяме и конструктора, като инициализираме нашия масив с някакъв начален капацитет, за да не се налага да го преоразмеряваме, когато добавим нов елемент. Нека разгледаме някои от типичните операции:

```
/**
 * Adds element to the list
 * @param item - the element you want to add
 */
public void add(Object item) {
   add(count, item);
}

/**
 * Inserts the specified element at given position in this list
 * @param index -
 *      index at which the specified element is to be inserted
 * @param item -
 *      element to be inserted
 * @throws IndexOutOfBoundsException
 */
public void add(int index, Object item) {
   if (index>count || index<0) {
      throw new IndexOutOfBoundsException(
         "Invalid index: " + index);
   }
   Object[] extendedArr = arr;
   if (count + 1 == arr.length) {
      extendedArr = new Object[arr.length * 2];
   }

   System.arraycopy(arr, 0, extendedArr, 0, index);
   count++;
   System.arraycopy(
      arr, index, extendedArr, index+1, count-index-1);
```

```
   extendedArr[index] = item;
   arr = extendedArr;
}
```

Реализирахме операцията добавяне на нов елемент, както и вмъкване на нов елемент. Тъй като едната операция е частен случай на другата, методът за добавяне вика този за вмъкване. Ако масивът ни се напълни заделяме два пъти повече място и копираме елементите от стария в новия масив.

Сигурно се чудите какви са тези коментари, оградени със звездички. Това са специални коментари в Java, които могат да се вграждат в програмата и обясняват какво прави всеки клас и всеки метод от класа. Те се наричат **Javadoc коментари**. Те могат да описва предназначението на методите, техните параметри, връщаната стойност и хвърляните изключения.

Нека се върнем към реализацията на списък чрез масив. Реализираме операциите търсене на елемент, намиране на елемент по индекс, и проверка за това дали даден елемент се съдържа в списъка:

```
/**
 * Returns the index of the first occurrence of the specified
 * element in this list.
 *
 * @param item - the element you are searching
 * @return the index of given element or -1 is not found
 */
public int indexOf(Object item) {
   if (item == null) {
      for (int i = 0; i < arr.length; i++) {
         if (arr[i] == null)
            return i;
      }
   } else {
      for (int i = 0; i < arr.length; i++)
         if (item.equals(arr[i]))
            return i;
   }
   return -1;
}

/**
 * Clears the list
 */
public void clear() {
   arr = new Object[0];
   count = 0;
}
```

```
/**
 * Checks if an element exists
 * @param item – the item to be checked
 * @return if the item exists
 */
public boolean contains(Object item) {
   int index = indexOf(item);
   boolean found = (index != -1);
   return found;
}

/**
 * @return the object on given position
 */
public Object elementAt(int index) {
   if (index>=count || index<0) {
      throw new IndexOutOfBoundsException(
         "Invalid index: " + index);
   }
   return arr[index];
}
```

Добавяме и операции за изтриване на елементи:

```
/**
 * Removes the element at the specified index
 * @param index - the index, whose element you want to remove
 * @return the removed element
 */
public Object remove(int index) {
   if (index>=count || index<0) {
      throw new IndexOutOfBoundsException(
         "Invalid index: " + index);
   }
   Object item = arr[index];
   System.arraycopy(arr, index+1, arr, index, count-index+1);
   arr[count - 1] = null;
   count--;
   return item;
}

/**
 * Removes the specified item and returns its index or -1
 * if item does not exists
 * @param item - the item you want to remove
*/
public int remove(Object item) {
```

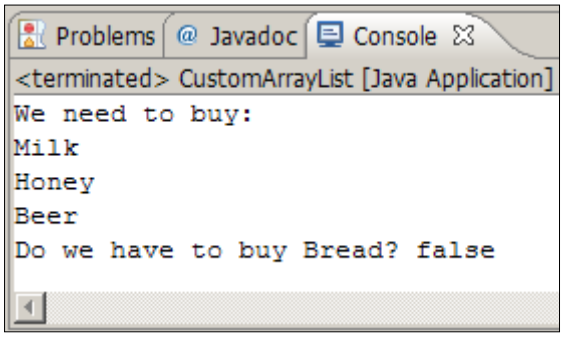
```
   int index = indexOf(item);
   if (index == -1) {
      return index;
   }
   System.arraycopy(arr, index+1, arr, index, count-index+1);
   count--;
   return index;
}
```

В горните методи премахваме елементи. За целта първо намираме търсения елемент, премахваме го, след което преместваме елементите след него, за да нямаме празно място на съответната позиция.

Нека сега видим как да използваме този наш клас. Добавяме и **main()** метод, в който ще демонстрираме някои от операциите. Да си направим списък с покупки и да го изведем на екрана. Освен това ще видим дали имаме да купуваме хляб и ще задраскаме маслините:

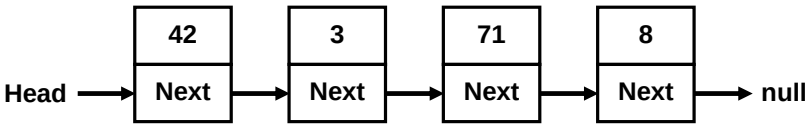
```
public static void main(String[] args) {
   CustomArrayList shoppingList = new CustomArrayList();
   shoppingList.add("Milk");
   shoppingList.add("Honey");
   shoppingList.add("Olives");
   shoppingList.add("Beer");
   shoppingList.remove("Olives");
   System.out.println("We need to buy:");
   for(int i=0; i<shoppingList.getLength(); i++) {
      System.out.println(shoppingList.elementAt(i));
   }
   System.out.println("Do we have to buy Bread? " +
      shoppingList.contains("Bread"));
}
```

Ето как изглежда изходът от програмата:



Свързан списък (динамична реализация)

Както видяхме, статичният списък има един сериозен недостатък – операциите добавяне и премахване от средата на списъка изискват преподреждане на елементите. При често добавяне и премахване (особено при голям брой елементи) това може да доведе до ниска производителност. В такива случаи се използват т. нар. свързани списъци. Разликата при тях е в структурата на елементите – докато при статичния списък елементите съдържат само конкретния обект, при динамичния списък елементите пазят информация за следващия елемент. Ето как изглежда свързаният списък в паметта:



За динамичната реализация ще са ни необходими два класа – класът **Node** – който ще представлява един отделен елемент от списъка и главният клас **DynamicList**:

```
/**
 * Represents dynamic list implementation
 * @author Tsvyatko Konov
 * @author Svetlin Nakov
 */
public class DynamicList {

   private class Node{
      Object element;
      Node next;

      Node(Object element, Node prevNode) {
         this.element = element;
         prevNode.next = this;
      }

      Node(Object element) {
         this.element = element;
         next = null;
      }
   }

   private Node head;
   private Node tail;
   private int count;
```

Първо виждаме помощния клас **Node** съдържащ указател към следващия елемент, както и поле за обекта, който пази. Както виждаме класът е

вложен в класа **DynamicList** и следователно може да се достъпва само от него. Отвън такъв клас не съществува. За нашия **DynamicList** създаваме три полета **head** – указател към началния елемент, **tail** – указател към последния елемент и **count** – брояч за елементите.

Създаваме си и конструктор:

```
public DynamicList() {
   this.head = null;
   this.tail = null;
   this.count = 0;
}
```

При първоначално конструиране списъкът е празен и затова **head** = **tail** = **null** и **count**=**0**.

Ще реализираме всички основни операции: добавяне и премахване на елементи, както и търсене на елемент.

Да започнем с операцията добавяне. Тя е относително проста:

```
/**
 * Add element at the end of the list
 * @param item - the element you want to add
 */
public void add(Object item) {
   if (head == null) {
      // We have empty list
      head = new Node(item);
      tail = head;
   } else {
      // We have non-empty list
      Node newNode = new Node(item, tail);
      tail = newNode;
   }
   count++;
}
```

Разглеждат се два случая: празен списък и непразен списък. И в двата случая се стремим да добавим елемента в края на списъка и след добавянето всички променливи (**head**, **tail** и **count** да имат коректни стойности).

Следва операцията изтриване по индекс. Тя е значително по-сложна от добавянето:

```
/**
 * Removes and returns element on the specific index
 * @param index - the index of the element you want to remove
```

```
 * @return the removed element
 * @exception IndexOutOfBoundsException - when index is invalid
 */
public Object remove(int index) {
   if (index>=count || index<0) {
      throw new IndexOutOfBoundsException(
         "Invalid index: " + index);
   }

   // Find the element at the specified index
   int currentIndex = 0;
   Node currentNode = head;
   Node prevNode = null;
   while (currentIndex < index) {
      prevNode = currentNode;
      currentNode = currentNode.next;
      currentIndex++;
   }

   // Remove the element
   count--;
   if (count==0) {
      head = null;
      tail = null;
   } else if (prevNode == null) {
      head = currentNode.next;
   } else {
      prevNode.next = currentNode.next;
   }
   return currentNode.element;
}
```

Първо се проверява дали посоченият за изтриване индекс съществува и ако не съществува се хвърля подходящо изключение. След това се намира елементът за изтриване чрез придвижване от началото на списъка към следващия елемент **index** на брой пъти. След като е намерен елементът за изтриване (currentNode), той се изтрива като се разглеждат 3 възможни случая:

- Списъкът остава празен след изтриването → изтриваме целия списък (**head** = **null**; **tail** = **null**).
- Елементът е в началото на списъка (няма предходен) → правим **head** да сочи елемента веднага след изтрития (или в частност към **null**, ако няма такъв).
- Елементът е в средата или в края на списъка → насочваме елемент преди него да сочи към елемента след него (или в частност към **null**, ако няма следващ).

Следва реализацията на изтриването на елемент по стойност:

```
/**
 * Removes the specified item and return its index
 * @param item – the item for removal
 * @return the index of the element or -1 if does not exist
 */
public int remove(Object item){
   // Find the element containing searched item
   int currentIndex = 0;
   Node currentNode = head;
   Node prevNode = null;
   while (currentNode != null) {
      if ((currentNode.element!=null &&
           currentNode.element.equals(item)) ||
         (currentNode.element==null) && (item==null)){
         break;
      }
      prevNode = currentNode;
      currentNode = currentNode.next;
      currentIndex++;
   }

   if (currentNode != null) {
      // Element is found in the list. Remove it
      count--;
      if (count==0) {
         head = null;
         tail = null;
      } else if (prevNode == null) {
         head = currentNode.next;
      } else {
         prevNode.next = currentNode.next;
      }
      return currentIndex;
   } else {
      // Element is not found in the list
      return -1;
   }
}
```

Изтриването по стойност на елемент работи като изтриването по индекс, но има 2 особености: търсеният елемент може и да не съществува и това налага допълнителна проверка; в списъка може да има елементи със стойност **null**, които трябва да предвидим и обработим по специален начин (вижте в кода). За да работи коректно изтриването, е необходимо елементите в масива да са сравними, т.е. да имат коректно реализиран модата **equals()** от **java.lang.Object**.

По-долу добавяме и операциите за търсене и проверка дали се съдържа даден елемент:

```
/**
 * Searches for given element in the list
 * @param item - the item you are searching for
 * @return the index of the first occurrence of
 * the element in the list or -1 when not found
 */
public int indexOf(Object item) {
   int index = 0;
   Node current = head;
   while (current != null) {
      if ((current.element!=null && current.element.equals(item))
            || (current.element==null) && (item==null)) {
         return index;
      }
      current = current.next;
      index++;
   }
   return -1;
}

/**
 * Check if the specified element exist in the list
 * @param item - the item you are searching for
 * @return true if the element exist or false otherwise
 */
public boolean contains(Object item) {
   int index = indexOf(item);
   boolean found = (index != -1);
   return found;
}
```

Търсенето на елемент работи както в метода за изтриване: започва се отначалото на списъка и се преравят последователно следващите един след друг елементи докато не се стигне до края на списъка.

Останаха още два сравнително лесни метода – достъп до елемент по индекс и извличане на дължината на списъка (броя елементи):

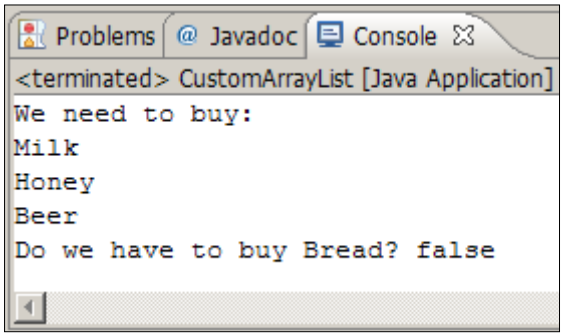
```
/**
 * @param index – the position of the element [0 … count-1]
 * @return the object at the specified index
 * @exception IndexOutOfBoundsException - when index is invalid
 */
public Object elementAt(int index) {
   if (index>=count || index<0) {
```

```
      throw new IndexOutOfBoundsException(
         "Invalid index: " + index);
   }
   Node currentNode = this.head;
   for (int i = 0; i < index; i++) {
      currentNode = currentNode.next;
   }
   return currentNode.element;
}

/**
 * @return the actual list length
 */
public int getLength() {
   return count;
}
```

Нека накрая видим нашия пример, този път реализиран чрез динамичен свързан списък.

```
public static void main(String[] args){
   DynamicList shoppingList = new DynamicList();
   shoppingList.add("Milk");
   shoppingList.add("Honey");
   shoppingList.add("Olives");
   shoppingList.add("Beer");
   shoppingList.remove("Olives");
   System.out.println("We need to buy:");
   for (int i=0; i<shoppingList.getLength(); i++) {
      System.out.println(shoppingList.elementAt(i));
   }
   System.out.println("Do we have to buy Bread? " +
      shoppingList.contains("Bread"));
}
```

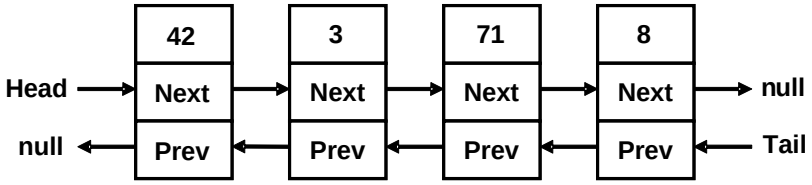
Както и очакваме, резултатът е същият както при реализацията на списък чрез масив:



Това показва, че можем да реализираме една и съща абстрактна структура от данни по фундаментално различни начини, но в крайна сметка ползвателите на структурата няма да забележат разлика в резултатите при използването ѝ. Единствената разлика ще е в скоростта на работа и в обема на заеманата памет.

Двойно свързани списъци

Съществува и т. нар. **двойно свързан списък** (двусвързан списък), при който всеки елемент съдържа стойността си и два указателя – към предходен и към следващ елемент (или `null`, ако няма такъв). Това ни позволява да обхождаме списъка, както напред така и в обратна посока. Това позволява някои операции да бъдат реализирани малко по-лесно. Ето как изглежда двусвързаният списък в паметта:



Класът ArrayList

След като се запознахме с някои от основните реализации на списъците, ще се спрем на класовете в Java, които ни предоставят списъчни структури "на готово". Първият от тях е класът `ArrayList`, който представлява динамично-разширяем масив. Той е реализиран по сходен начин със статичната реализация на списък, която разгледахме по-горе. Имаме възможност да добавяме, премахваме и търсим елементи. Някои по-важни методи, които можем да използваме са:

- `add(Object)` – добавяне на нов елемент
- `add(index, Object)` – добавяне елемент на определено място (индекс)
- `size()` – връща броя на елементите в списъка
- `remove(Object)` – премахване определен елемент

- **remove(index)** – премахване на елемента на определено място (индекс)
- **clear()** – изчистване на списъка

Както видяхме, един от основните проблеми при тази реализация е преоразмеряването на вътрешния масив при добавянето и премахването на елементи. В класа **ArrayList** проблемът е решен чрез предварително създаване на по-голям масив, който ни предоставя възможност да добавяме елементи, без да преоразмеряваме масива при всяко добавяне или премахване на елементи. След малко ще обясним това в детайли.

Класът ArrayList – пример

В класа **ArrayList** можем да записваме всякакви елементи – числа, символни низове и други обекти. Ето един малък пример:

```
import java.util.ArrayList;
import java.util.Date;

public class ArrayListExample {
   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("Hello");
      list.add(5);
      list.add(3.14159);
      list.add(new Date());

      for (int i=0; i<list.size(); i++) {
         Object value = list.get(i);
         System.out.printf("Index=%d; Value=%s\n", i, value);
      }
   }
}
```

В примера създаваме **ArrayList** и записваме в него няколко елемента от различни типове: **String**, **int**, **double** и **Date**. След това итерираме по елементите и ги отпечатваме. Ако изпълним примера, ще получим следния резултат:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=Sat Nov 29 23:17:01 EET 2008
```

ArrayList с числа – пример

Ако искаме да си направим масив от числа и след това да обработим числата, примерно да намерим тяхната сума, се налага да преобразуваме

типа **Object** към число, защото **ArrayList** не може да пази числа и пази винаги обекти. Ето примерен код, който сумира елементите на **ArrayList**:

```
ArrayList list = new ArrayList();
list.add(2);
list.add(3);
list.add(4);
int sum = 0;
for (int i=0; i<list.size(); i++) {
   Integer value = (Integer) list.get(i);
   sum = sum + value.intValue();
}
System.out.println("Sum = " + sum);
// Output: Sum = 9
```

Ако пуснете горния пример в Eclipse, ще получите множество забележки, идващи от компилатора, които ви напомнят, че ползвате непараметризиран тип на данните, което не е добра практика. Преди да ви покажем още примери за работа с класа **ArrayList** ще да ви запознаем с една концепция в Java, наречена "шаблонни типове данни". Тя дава възможност да се параметризират списъците и колекциите в Java и улеснява значително работата с тях.

Шаблонни класове (generics)

Когато използваме класа **ArrayList**, а и всички класове, имплементиращи интерфейса **java.util.List,** се сблъскваме с проблема, който видяхме по-горе: когато добавяме нов елемент от даден клас ние го предаваме като обект от тип **Object**. Когато по-късно търсим даден елемент, ние го получаваме като **Object** и се налага да го превърнем в изходния тип. Не ни се гарантира, обаче, че всички елементи в списъка ще бъдат от един и същ тип. Освен това превръщането от един тип в друг отнема време, което забавя излишно изпълнението на програмата.

За справяне в описаните проблеми на помощ идват шаблонните класове. Образно казано те са шаблони създадени да работят с един или няколко типа, като при създаването си ние указваме какъв точно тип обекти ще съхраняваме в тях. Създаването на инстанция от даден шаблонен тип, примерно **GenericType**, става като в счупени скоби се зададе типа, от който трябва да бъдат елементите му:

```
GenericType<T> instance = new GenericType<T>();
```

Този тип **T** може да бъде всеки наследник на класа **java.lang.Object**, примерно **String** или **Date**. Понеже числата не са обекти и не наследяват класа **Object**, ако трябва да ги използваме като тип в шаблонен клас, трябва да използваме съответния им обвиващ (wrapper) клас. Така вместо

примитивния тип **int** трябва да ползваме класа **Integer**, а вместо типа **boolean** трябва да ползваме класа **Boolean**. Ето няколко примера:

```
ArrayList<Integer> intList = new ArrayList<Integer>();
ArrayList<Boolean> boolList = new ArrayList<Boolean>();
ArrayList<Double> realNumbersList = new ArrayList<Double>();
```

Нека сега разгледаме някои от шаблонните колекции в Java.

Класът ArrayList<T>

ArrayList<T> е шаблонният вариант на **ArrayList**. При инициализацията на обект от тип **ArrayList<T>** указваме типа на елементите, който ще съдържа списъка, т. е. заместваме означения с **T** тип с някой истински тип данни (например число или стринг).

Например искаме да създадем списък от целочислени елементи. Можем да го направим по следния начин:

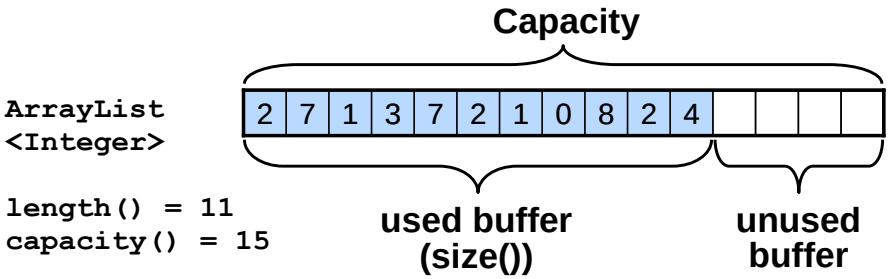
```
ArrayList<Integer> genericList = new ArrayList<Integer>();
```

Създаденият по този начин списък може да приема като стойности цели числа, но не може и други обекти, например символни низове. Ако се опитаме да добавим към **ArrayList<Integer>** обект от тип **String**, ще получим грешка по време на компилация. Чрез шаблонните типове компилаторът на Java ни пази от грешки при работа с колекции.

Забележете, че не посочваме типа **int**, а неговия обвиващ тип **Integer**. Причината за това е фактът, че шаблоните приемат като параметър само референтни типове (обекти) и не могат да работят с обикновени стойностни типове, които не се пазят в динамичната памет. По тази причина ползването на **ArrayList<String>** става директно, а ползването на списък от **int** или **double** изисква да ползваме съответните обвиващи типове: **ArrayList<Integer>** и **ArrayList<Double>**.

Класът ArrayList – представяне чрез масив

Класът **ArrayList** се представя в паметта като масив, от който една част съхранява елементите му, а останалите са свободни и се пазят като резервни. Благодарение на резервните празни елементи в масива операцията добавяне почти винаги успява да добави новия елемент без да разширява (преоразмерява) масива. Понякога, разбира се, се налага преоразмеряване, но понеже всяко преоразмеряване увдвоява размера на масива, това се случва толкова рядко, че може да се пренебрегне на фона на броя добавяния. Можем да си представим един **ArrayList** като масив, който има някакъв капацитет и запълненост до определено ниво:



Благодарение на резервното пространство в масива, съхраняващ елементите на класа **ArrayList<T>**, той е изключително удобна структура от данни, когато е необходимо бързо добавяне на елементи, извличане на всички елементи и пряк достъп до даден елемент по индекс.

Може да се каже, че **ArrayList<T>** съчетава добрите страни на списъците и масивите – бързо добавяне, променлив размер и директен достъп по индекс.

Класът ArrayList – истинско представяне в паметта

Ако трябва да сме точни, трябва да отбележим, че горната картинка всъщност не отразява точното представяне на класа **ArrayList<T>** в паметта. Причината за това е, че в Java няма истински шаблонни типове, а само имитация на такива. Всички шаблонни типове **T** изчезват още по време на компилация и се преобразуват в **Object**, т. е. долните три дефиниции след компилация стават абсолютно еднакви:

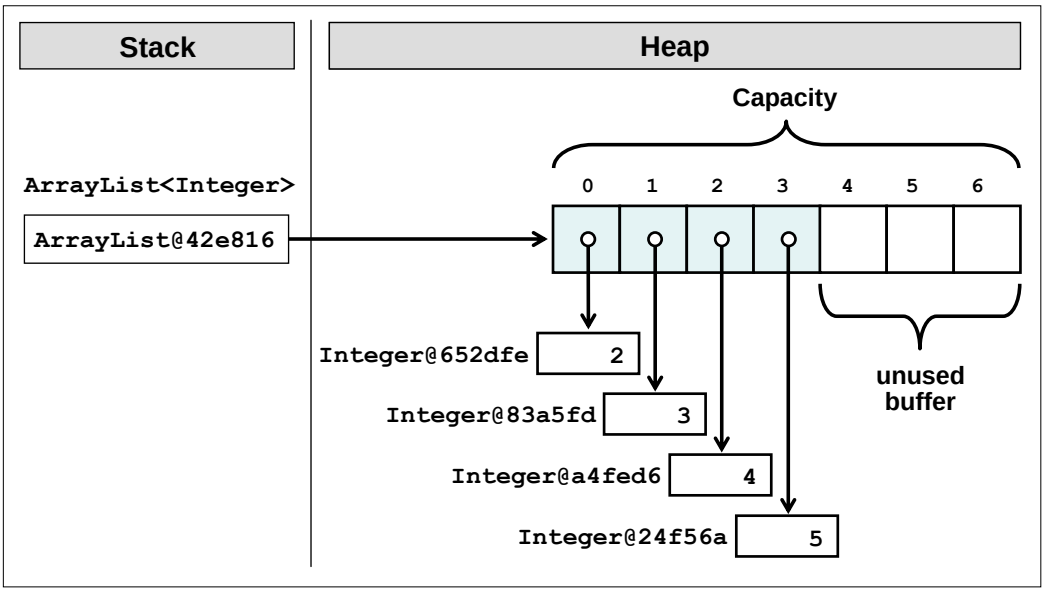
```
ArrayList<Integer> genericList = new ArrayList<Integer>();
ArrayList<Object> objList = new ArrayList<Object>();
ArrayList plainList = new ArrayList ();
```

Всички параметризирани колекции в Java са всъщност колекции от обекти и по тази причина работят по-бавно, отколкото масивите от примитивни типове (например **int[]**). При записването на нов елемент от примитивен тип в **ArrayList** той се премества в динамичната памет. При достъп до елемент от **ArrayList**, той се връща като обект и след това може да бъде преобразуван обратно към примитивен тип (например към число).

Нека сме изпълнили следния код:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(2);
list.add(3);
list.add(4);
list.add(5);
```

Истинската картинка, която отразява представянето на **ArrayList** в паметта е следната:



Вижда се, че всяка отделна стойност на списъка от числа е обект от тип **Integer**, разположен в динамичната памет (heap). В масива се пазят не самите стойности на елементите, а техните адреси (указатели). По тази причина достъпът до елементите на **ArrayList<Integer>** е по-бавен, отколкото достъпът до елементите на **int[]**.

Кога да използваме ArrayList<T>?

Както вече обяснихме, класът **ArrayList<T>** използва вътрешно масив за съхранение на елементите, който удвоява размера си, когато се препълни. Тази негова специфика води до следните особености:

- Търсенето по индекс става много бързо – можем да достъпваме с еднаква скорост всеки един от елементите независимо от общия им брой.
- Търсенето по стойност на елемент работи с толкова сравнения, колкото са елементите, т.е. не е бързо.
- Добавянето и премахването на елементи е бавна операция – когато добавяме или премахваме елементи, особено, ако те не се намират в края на списъка, се налага да разместваме всички останали елементи, а това е много бавна операция.
- При добавяне понякога се налага и увеличаване на капацитета на масива, което само по себе си е бавна операция, но се случва много рядко и средната скорост на добавяне на елемент към **ArrayList** не зависи от броя елементи, т.е. работи много бързо.

Използвайте ArrayList<T>, когато не очаквате често вмъкване и премахване на елементи, но очаквате да добавяте нови елементи в края или ползвате елементите по индекс.

Прости числа в даден интервал – пример

След като се запознахме отвътре с реализацията на структурата списък и класа **ArrayList<T>**, нека видим как да използваме този клас. Ще разгледаме проблема за намиране на простите числа в някакъв интервал. За целта ще използваме следния алгоритъм:

```
public static ArrayList<Integer> getPrimes(int start, int end) {
   ArrayList<Integer> primesList = new ArrayList<Integer>();
   for (int num = start; num <= end; num++) {
      boolean prime = true;
      for (int div = 2; div <= Math.sqrt(num); div++) {
         if (num % div == 0) {
            prime = false;
            break;
         }
      }
      if (prime)
         primesList.add(num);
   }
   return primesList;
}

public static void main(String[] args) {
   ArrayList<Integer> primes = getPrimes(200, 300);
   for (int p : primes) {
      System.out.printf("%d ", p);
   }
   System.out.println();
}
```

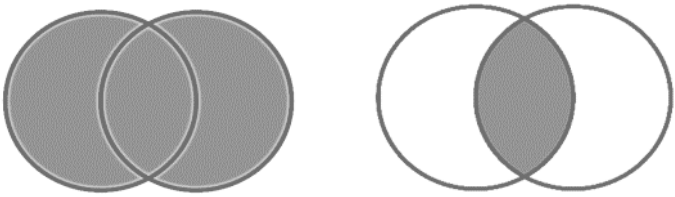
От математиката знаем, че ако едно число не е просто, то съществува поне един делител в интервала [2 …к орен квадратен от даденото число]. Точно това използваме в примера по-горе. За всяко число търсим дали има делител в този интервал. Ако срещнем, то числото не е просто и можем да продължим със следващото. Постепенно чрез добавяне на прости числа пълним списъка, след което го обхождаме и го извеждаме на екрана. Ето го и изходът от горния код:

```
Problems  Javadoc  Console
<terminated> Primes [Java Application] C:\Program Files\Java\jdk1.6.0_07\bin\javaw.exe (09
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
```

Обединение и сечение на списъци – пример

Нека сега разгледаме един по-интересен пример. Имаме за цел да напишем програма, която може да намира обединенията и сеченията на две множества числа.

Обединение **Сечение**



Можем да приемем, че имаме два списъка и искаме да вземем елементите, които се намират и в двата едновременно (сечение) или търсим тези, които се намират поне в единия от двата (обединение).

Нека разгледаме едно възможно решение на задачата:

```
public static ArrayList<Integer> union(ArrayList<Integer>
     firstList, ArrayList<Integer> secondList) {
   ArrayList<Integer> union = new ArrayList<Integer>();
     union. addAll(firstList);

   for (Integer item : secondList) {
     if (!union.contains(item)) {
       union.add(item);
     }
   }
   return union;
}

public static ArrayList<Integer> intersect(ArrayList<Integer>
     firstList, ArrayList<Integer> secondList) {
   ArrayList<Integer> intersect = new ArrayList<Integer>();
   for (Integer item : firstList) {
     if (!secondList.contains(item)) {
       intersect.add(item);
     }
   }
   return intersect;
```

```
}

public static void printList(ArrayList<Integer> list) {
   System.out.print("{ ");
   for (Integer item : list) {
      System.out.print(item);
      System.out.print(" ");
   }
   System.out.println("}");
}

public static void main(String[] args) {
   ArrayList<Integer> firstList = new ArrayList<Integer>();
   firstList.add(1);
   firstList.add(2);
   firstList.add(3);
   firstList.add(4);
   firstList.add(5);
   System.out.print("firstList = ");
   printList(firstList);

   ArrayList<Integer> secondList = new ArrayList<Integer>();
   secondList.add(2);
   secondList.add(4);
   secondList.add(6);
   System.out.print("secondList = ");
   printList(secondList);

   ArrayList<Integer> unionList = union(firstList, secondList);
   System.out.print("union = ");
   printList(unionList);

   ArrayList<Integer> intersectList =
      intersect(firstList, secondList);
   System.out.print("intersect = ");
   printList(intersectList);
}
```

Програмната логика в това решение директно следва определенията за обединение и сечение на множества. Ползваме операциите търсене на елемент в списък и добавяне на елемент към списък.

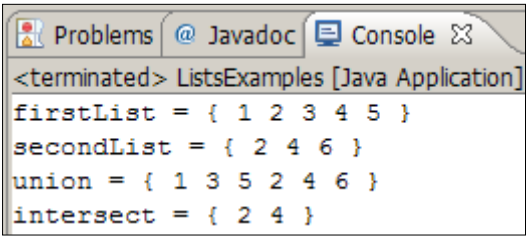
Ще решим проблема по още един начин: като използваме методите **addAll(Collection c)** и **retainAll(Collection c)** от интерфейса **java.util.Collection**, който **ArrayList** имплементира:

```
public static void main(String[] args) {
   ArrayList<Integer> firstList = new ArrayList<Integer>();
```

```
    firstList.add(1);
    firstList.add(2);
    firstList.add(3);
    firstList.add(4);
    firstList.add(5);
    System.out.print("firstList = ");
    printList(firstList);

    ArrayList<Integer> secondList = new ArrayList<Integer>();
    secondList.add(2);
    secondList.add(4);
    secondList.add(6);
    System.out.print("secondList = ");
    printList(secondList);

    ArrayList<Integer> unionList = new ArrayList<Integer>();
    unionList.addAll(firstList);
    unionList.removeAll(secondList);
    unionList.addAll(secondList);
    System.out.print("union = ");
    printList(unionList);

    ArrayList<Integer> intersectList = new ArrayList<Integer>();
    intersectList.addAll(firstList);
    intersectList.retainAll(secondList);
    System.out.print("intersect = ");
    printList(intersectList);
}
```

За да направим сечение правим следното: слагаме всички елементи от първия списък, след което премахваме всички елементи, които не се съдържат във втория (чрез **retainAll()**). Обединението правим като добавим елементите от първия списък, след което премахнем всички които се съдържат и в двата (чрез **removeAll()**), след което добавяме всички елементи от втория списък.

Резултатът и от двете програми изглежда по един и същ начин:



Превръщане на ArrayList в масив и обратното

Тъй като класът **ArrayList<T>** и масивите много си приличат, често се налага да преобразуваме от **ArrayList<T>** към масив **T[]** и обратното. За преобразуването на масив към **ArrayList** няма стандартен метод или конструктор. За обратното преобразование има системен метод **toArray()**, но той има някои особености: изисква да му се подаде като параметър резултатният масив и след това да се извърши преобразование на върнатата стойност. Нека видим тези преобразования в действие:

ArrayConversions.java

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayConversions {
   public static void main(String[] args) {
      int[] arr = new int[] {1, 2, 3};

      // Convert the array to ArrayList
      ArrayList<Integer> list =
         new ArrayList<Integer>(arr.length);
      for (int i=0; i<arr.length; i++) {
         list.add(arr[i]);
      }

      // Append new number to the ArrayList
      list.add(4);

      // Convert the ArrayList to array
      Integer[] ints =
          (Integer[]) list.toArray(new Integer[list.size()]);

      // Print the array
      System.out.println(Arrays.toString(ints));
   }
}
```

Ако изпълним програмата, ще получим следния резултат:

```
[1, 2, 3, 4]
```

За съжаление в Java шаблонните класове и масивите не са измислени добре (както е направено в .NET, JavaScript и PHP) и поради това нямаме директна съвместимост между **ArrayList** и масив. Остава ни да се надяваме някой ден на Sun и на Java обществото да им дойде акъла и да поправят грешката си. До тогава ще преминаваме междумасиви и списъци ръчно, както в примера по-горе.

Класът LinkedList<T>

Този клас представлява динамична реализация на двусвързан списък. Елементите му пазят информация за обекта, който съхраняват, и указател към следващия и предишния елемент.

Кога да използваме LinkedList<T>?

Видяхме, че динамичната и статичните реализации имат специфика по отношение бързодействие на различните операции. С оглед на структурата на свързания списък трябва да имаме предвид следното:

- Добавянето на елементи в **`LinkedList`** става много бързо – независимо от броя на елементите.
- Можем да добавяме бързо в началото и в края на списъка (за разлика от **`ArrayList<T>`**).
- Търсенето на елемент по индекс или по съдържание в **`LinkedList`** е бавна операция, тъй като се налага да обхождаме всички елементи последователно като започнем от началото на списъка.
- Изтриването на елемент е бавна операция, защото включва търсене.

Основни операции в класа LinkedList<T>

`LinkedList<T>` притежава същите операции като **`ArrayList<T>`**, което прави двата класа взаимозаменяеми в зависимост от конкретната задача. По-късно ще видим, че **`LinkedList<T>`** се използва и при работа с опашки.

Кога да ползваме LinkedList<T>?

Като цяло класът **`LinkedList<T>`** се използва много рядко, защото **`ArrayList<T>`** върши същата работа не по-бавно, а предлага в допълнение и други бързи операции.

Стек

Да си представим няколко кубчета, които сме наредили едно върху друго. Можем да слагаме ново кубче на върха, както и да махаме най-горното кубче. Или да си представим една ракла. За да извадим прибраните дрехи или завивки от дъното на раклата, трябва първо да махнем всичко, което е върху тях.

Точно тази конструкция представлява стекът – можем да добавяме елементи и да извличаме последният добавен елемент, но не и предходните (които са затрупани под него). Стекът е често срещана и използвана структура от данни. Стек се използва и вътрешно от Java виртуалната машина за съхранение на променливите в програмата и параметрите при извикване на метод.

Абстрактна структура данни "стек"

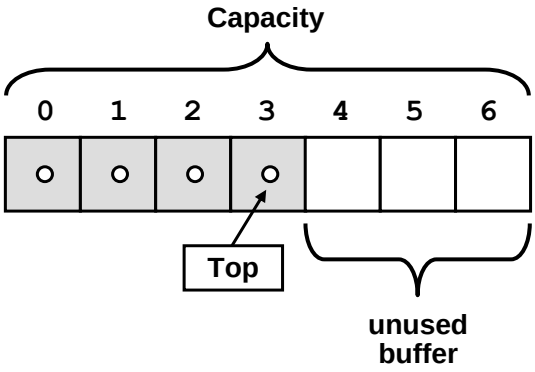
Стекът представлява структура от данни с поведение "последният влязъл първи излиза". Както видяхме в примера с кубчетата, елементите могат да се добавят и премахват само от върха на стека.

Структурата от данни стек също може да има различни реализации, но ние ще се спрем на двете основни – динамичната и статичната реализация.

Статичен стек (реализация с масив)

Както и при статичния списък и можем да използваме масив за пазене на елементите на стека. Можем да имаме индекс или указател, който сочи към елемента, който се намира на върха. Обикновено при запълване на масива следва заделяне на двойно повече памет, както това се случва при статичния списък (`ArrayList`).

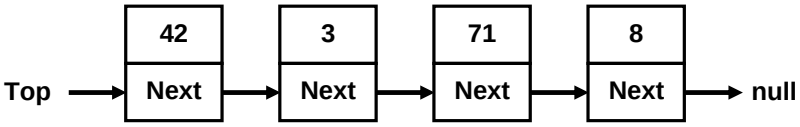
Ето как можем да си представим един статичен стек:



Както и при статичния масив се поддържа свободна буферна памет с цел по-бързо добавяне.

Свързан стек (динамична реализация)

За динамичната реализация ще използваме елементи, които пазят, освен обекта, и указател към елемента, който се намира "по-долу". Тази реализация решава ограниченията, които има статичната реализация както и необходимостта от разширяване на масива при нужда:



Когато стекът е празен, върхът има стойност `null`. При добавяне на нов елемент, той се добавя на мястото, където сочи върхът, след което върхът се насочва към новия елемент. Премахването става по аналогичен начин.

Класът Stack<T>

В Java можем да използваме класа **java.util.Stack<T>**, предоставя структурата от данни стек. Използвана е статичната имплементация като вътрешният масив се преоразмерява при необходимост.

Класът Stack<T> – основни операции

Реализирани са всички необходими операции за работа със стек:

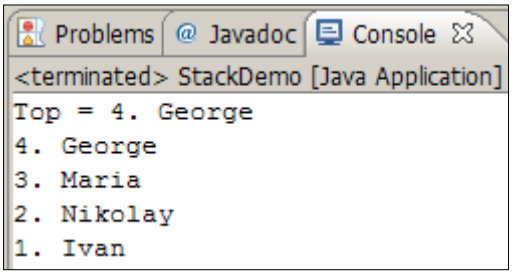
- **push(T) –** позволява ни добавянето на нов елемент на върха на стека
- **pop() –** връща ни най-горния елемент като го премахва от стека
- **peek() –** връща най горния елемент без да го премахва
- **size() –** връща броя на елементите в стека
- **clear() –** премахва всички елементи
- **contains(T) –** проверява дали елемента се съдържа в стека
- **toArray() –** връща масив, съдържащ елементите от стека

Използване на стек – пример

Нека сега видим един прост пример как да използваме стек. Ще добавим няколко елемента, след което ще ги вземе всички и ще ги изведем на конзолата.

```
public static void main(String[] args) {
   Stack<String> stack = new Stack<String>();
   stack.push("1. Ivan");
   stack.push("2. Nikolay");
   stack.push("3. Maria");
   stack.push("4. George");
   System.out.println("Top = " + stack.peek());
   while (stack.size() > 0) {
      String personName = stack.pop();
      System.out.println(personName);
   }
}
```

Тъй като стекът е структура "последен влязъл – пръв излязъл", програмата ще изведе записите в ред обратен на реда на добавянето. Ето как нейният изход:



Проверка за съответстващи скоби – пример

Да разгледаме следната задача: имаме числов израз, на който искаме да проверим дали броят на отварящите скоби е равен на броя на затварящите. Спецификата на стека ни позволява да проверяваме дали скобата, която сме срещнали има съответстваща затваряща. Когато срещнем отваряща, я добавяме към стека. При срещане на затваряща вадим елемент от стека. Ако стекът остане празен преди края на програмата, в момент, в който трябва да извадим още един елемент, значи скобите са некоректно поставени. Същото важи и ако накрая в стека останат някакви елементи. Ето една примерна реализация:

```
public static void main(String[] args) {
   String expression =
     "1 + (3 + 2 - (2+3) * 4 - ((3+1)*(4-2)))";
   Stack<Integer> stack = new Stack<Integer>();
   boolean correctBrackets=true;

   for (int index = 0; index < expression.length(); index++) {
      char ch = expression.charAt(index);
      if (ch == '(') {
         stack.push(index);
      } else if (ch == ')') {
         if(stack.isEmpty()){
            correctBrackets=false;
            break;
         }
         stack.pop();
      }
   }
   if(!stack.isEmpty())
      correctBrackets=false;
   System.out.println("Are the brackets correct? " +
      correctBrackets);
}
```

Ето как изглежда изходът е примерната програма:

```
Are the brackets correct? true
```

Опашка

Структурата "**опашка**" е създадена да моделира опашки, като например опашка от чакащи документи за принтиране, чакащи процеси за достъп до общ ресурс и други. Такива опашки много удобно и естествено се моделират чрез структурата "опашка". В опашките можем да добавяме елементи само най-отзад и да извличаме елементи само от най-отпред.

Нека, например, искаме да си купим билет за концерт. Ако отидем по-рано ще си купим едни от билети. Ако обаче се забавим ще трябва да се наредим на опашката и да изчакаме всички желаещи преди нас да си купят билети. Нямаме право да се прередим, защото охраната ще ни се скара. Това поведение е аналогично за обектите в АТД опашка.

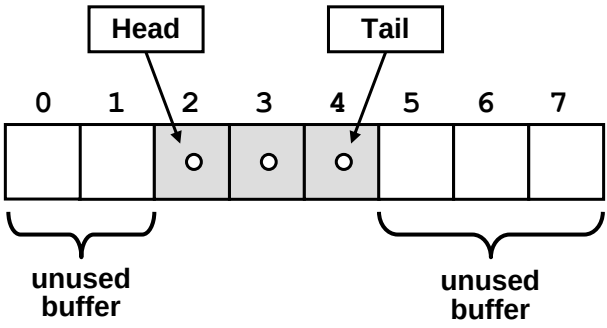
Абстрактна структура данни "опашка"

Абстрактната структура опашка изпълнява условието "първият влязъл първи излиза". Добавените елементи се нареждат в края на опашката, а при извличане поредният елемент се взима от началото (главата) ѝ.

Както и при списъка за структурата от данни опашка отново е възможна статична и динамична реализация.

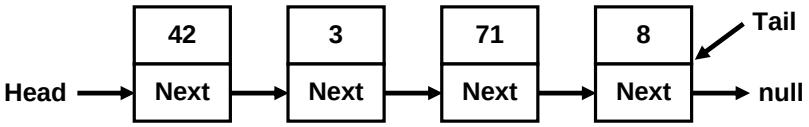
Статична опашка (реализация с масив)

В статичната опашка отново ще използваме масив за пазене на данните. При добавяне на елемент той се добавя на индекса, който следва края, след което края започва да сочи към ново добавения елемент. При премахване на елемент, се взима елемента, към който сочи главата, след което главата започва да сочи към следващия елемент. По този начин опашката се придвижва към края на масива. Когато стигне до края, при добавяне на нов елемент той се добавя на първо място. Ето защо тази имплементация се нарича още **зациклена опашка**, тъй като мислено залепяме началото и края на масива и опашката обикаля в него:



Свързана опашка (динамична реализация)

Динамичната реализация на опашката много прилича на тази на свързания списък. Елементите отново съдържат две части – обекта и указател към предишния елемент:



Тук обаче елементите се добавят на опашката, а се вземат от главата, като нямаме право да взимаме или добавяме елементи на друго място.

Интерфейсът Queue<T>

В Java се използва динамичната реализация на опашка чрез интерфейса **`Queue<T>`.** Както видяхме, интерфейсите декларират определени методи и свойства (т. е. АТД). При инициализация ние използваме класа **`LinkedList<T>`,** като му указваме да има поведение на опашка, т.е. получаваме имплементация със свързани елементи, която ще притежава методите на характерни за опашка. Тук отново можем да укажем типа на елементите, с които ще работим, тъй като опашката и свързаният списък са шаблонни типове.

Интерфейсът Queue<T> – основни операции

Queue<T> ни предоставя основните операции характерни за структурата опашка. Ето някои от често използваните:

- **`offer(T)`** – добавя елемент накрая на опашката
- **`poll()`** – взима елемента от началото на опашката и го премахва
- **`peek()`** – връща елементът от началото на опашката без да го премахва
- **`clear()`** – премахва всички елементи от опашката
- **`contains(T)`** – проверява дали елемента се съдържа в опашката

Използване на опашка – пример

Нека сега разгледаме прост пример. Да си създадем една опашка и добавим в нея няколко елемента. След това ще извлечем всички чакащи елементи и ще ги изведем на конзолата:

```
public static void main(String[] args) {
   Queue<String> queue = new LinkedList<String>();
   queue.offer("Message One");
   queue.offer("Message Two");
   queue.offer("Message Three");
   queue.offer("Message Four");
```

```
   while (queue.size() > 0) {
      String msg = queue.poll();
      System.out.println(msg);
   }
}
```

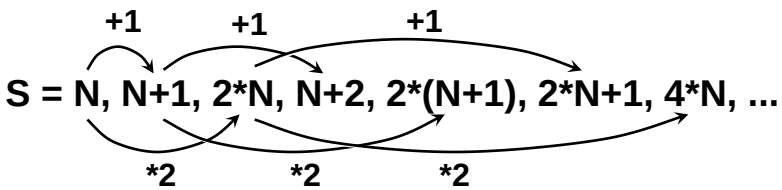
Ето как изглежда изходът е примерната програма:

```
Message One
Message Two
Message Three
Message Four
```

Вижда се, че елементите излизат от опашката в реда, в който са постъпили в нея.

Редицата N, N+1, 2*N – пример

Нека сега разгледаме задача, в която използването на структурата опашка ще бъде много полезна за реализацията. Да вземем редицата числа, чиито членове се поличават по-следния начин: първият елемент е N; вторият получаваме като съберем N с 1; третият – като умножим първия с 2 и така последователно умножаваме всеки елемент с 2 и го добавяме накрая на редицата, след което го събираме с 1 и отново го поставяме накрая на редицата. Можем да илюстрираме този процес със следната фигура:



Както виждаме, процесът се състои във взимане на елементи от началото на опашка и поставянето на други в края й. Нека сега видим примерна реализация, в която N=3 и търсим номера на член със стойност 16:

```
public static void main(String[] args) {
   int n = 3;
   int p = 16;

   Queue<Integer> queue = new LinkedList<Integer>();
   queue.offer(n);
   int index = 0;
   System.out.print("S =");
   while (queue.size() > 0) {
      index++;
```

```
      int current = queue.poll();
      System.out.print(" " + current);
      if (current == p) {
         System.out.println();
         System.out.println("Index = " + index);
         return;
      }
      queue.offer(current + 1);
      queue.offer(2 * current);
   }
}
```

Ето как изглежда изходът е примерната програма:

```
S = 3 4 6 5 8 7 12 6 10 9 16
Index = 11
```

Както видяхме, стекът и опашката са две специфични структури с определени правила за достъпа до елементите в тях. Опашка използваме, когато очакваме да получим елементите в реда, в който сме ги поставили, а стек – когото елементите ни трябват в обратен ред.

Упражнения

1. Реализирайте структурата двойно свързан динамичен списък – списък, чиито елементи имат указател, както към следващия така и към предхождащия го елемент. Реализирайте операциите добавяне, премахване и търсене на елемент, добавяне на елемент на определено място (индекс), извличане на елемент по индекс и метод, който връща масив с елементите на списъка.

2. Създайте клас **DynamicStack** представляващ динамична реализация на стек. Добавете методи за необходимите операции.

3. Реализирайте структурата дек. Това е специфична структура, позволяваща елементи да бъдат добавяни и премахвани от двата й края. Нека освен това, елемент поставен от едната страна да може да бъде премахнат само от същата. Реализирайте операции за премахване добавяне и изчистване на дека. При невалидна операция подавайте подходящо изключение.

4. Реализирайте структурата "зациклена опашка" с масив, който при нужда удвоява размера си. Имплементирайте необходимите методи за добавяне към опашката, извличане на елемента, който е наред и поглеждане на елемента, който е наред, без да го премахвате от опашката. При невалидна операция подавайте подходящо изключение.

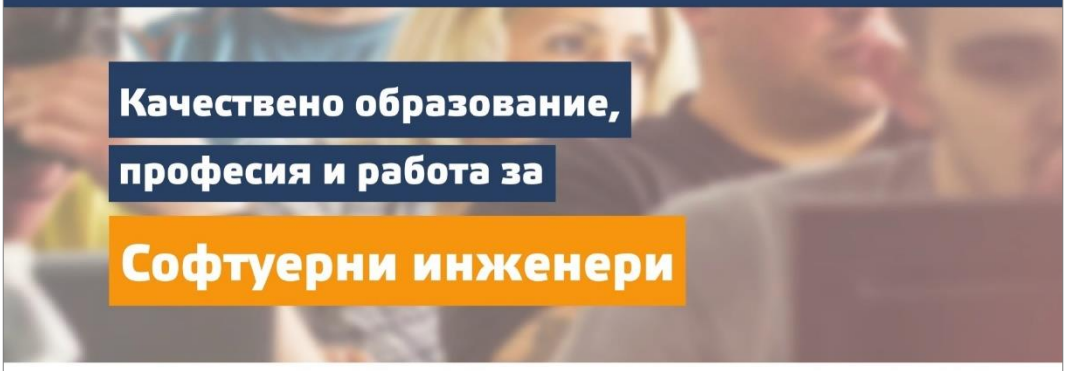
5. Реализирайте сортиране на числа в динамичен свързан списък, без да използвате допълнителен масив.

6. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в ширина" – Breadth-First-Search (BFS) – може да намерите стотици статии за него в Интернет.
7. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в дълбочина" – Depth-First-Search (DFS) – може да намерите стотици статии за него в Интернет.

Решения и упътвания

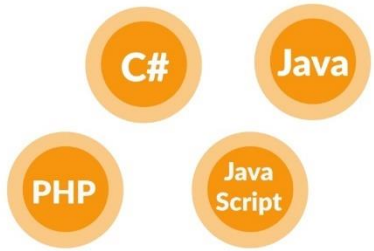
1. Вижте динамичната реализация на едносвързан списък, която разгледахме в секцията "Свързан списък".
2. Вижте динамичната реализация на едносвързан списък, която разгледахме в секцията "Свързан списък".
3. Използвайте два стека с общо дъно. По този начин, ако добавяме елементи отляво на дека ще влизат в левия стек, след което ще могат да бъдат премахнати отново оттам. Аналогично за десния стек.
4. Използвайте масив. Когато стигнем до последния индекс ще добавим следващия елемент в началото на масива. За точното пресмятане на индексите използвайте остатък от делене на дължината на масива. При нужда от преоразмеряване на масива можете да го направите по аналогия с реализираното преоразмеряване в секцията "Статичен списък".
5. Използвайте просто сортиране по метода на мехурчето. Започваме от най левия елемент, като проверяваме дали е по-малък от следващия. Ако не, им сменяме местата. После сравняваме със следващия и т.н. докато достигнем до такъв, който е по-голям от него или не стигнем края на масива. Връщаме се в началото и взимаме пак първия като повтаряме процедурата. Ако първият е по-малък, взимаме следващия и започваме да сравняваме. Повтаряме тези операции докато не стигнем до момент, в който сме взели последователно всички елементи и на нито един не се наложило да бъде преместен.
6. Алгоритъмът е много лесен: започваме от празна опашка, в която слагаме коренната директория (от която стартира обхождането). След това докато опашката не остане празна, изваждаме от нея поредната директория, отпечатваме я и прибавяме към опашката всички нейни поддиректории. По този начин ще обходим файловата система в ширина. Ако в нея няма цикли (както е под Windows), процесът ще е краен.
7. Ако в решението на предната задача заместим опашката със стек, ще получим обхождане в дълбочина. Хитро, нали?

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 17. Дървета и графи

Автор

Веселин Колев

В тази тема...

В настоящата тема ще разгледаме т. нар. дървовидни структури от данни, каквито са дърветата и графите. Познаването на свойствата на тези структури е важно за съвременното програмиране. Всяка от тях се използва за моделирането на проблеми от реалността, които се решават ефективно с тяхна помощ. Ще разгледаме в детайли какво представляват дървовидните структури данни и ще покажем техните основни предимства и недостатъци. Ще дадем примерни реализации и задачи, демонстриращи реалната им употреба. Ще се спрем по-подробно на двоичните дървета, наредените двоични дървета за претърсване и балансираните дървета. Ще разгледаме структурата от данни "граф", видовете графи и тяхната употреба. Ще покажем как се работи с вградените в Java платформата имплементации на балансирани дървета.

Дървовидни структури

В много ситуации в ежедневието се налага да опишем (моделираме) съвкупност от обекти, които са взаимно свързани по някакъв начин и то така, че не могат да бъдат описани чрез досега изложените линейни структури от данни. В следващите няколко точки от тази тема ще дадем примери за такива структури, ще покажем техните свойства и съответно практическите задачи, които са довели до тяхното възникване.

Дървета

В програмирането дърветата са изключително често използвана структура от данни, защото те моделират по естествен начин всякакви йерархии от обекти, които постоянно ни заобикалят в реалния свят. Нека дадем един пример, преди да изложим терминологията, свързана с дърветата.

Пример – йерархия на участниците в един софтуерен проект

Да вземем за пример един екип, отговорен за изработването на даден софтуерен проект. Участниците в него са взаимно свързани с връзката

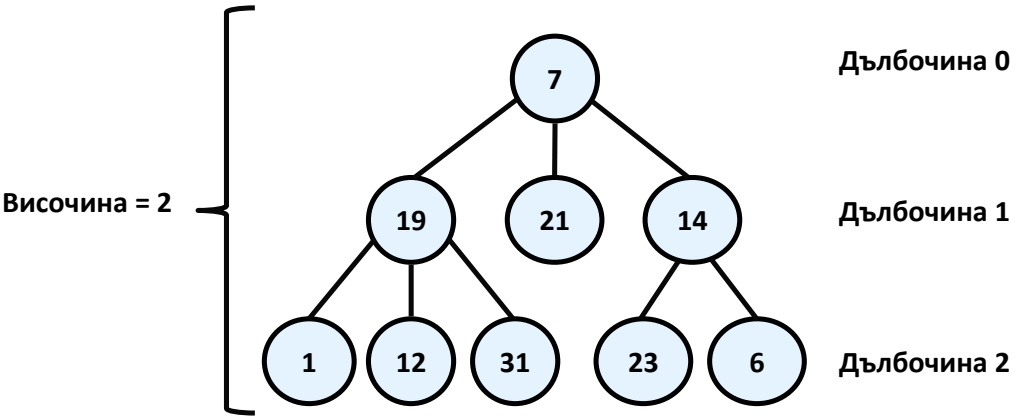
ръководител-подчинен. Ще разгледаме една конкретна ситуация, в която имаме екип от 9 души:

Каква информация можем да извлечем от така изобразената йерархия? Прекият шеф на програмистите е съответно "Ръководител програмисти". "Ръководител проект" е също е техен началник, но непряк, т.е. те отново са му подчинени. "Ръководител програмисти" е подчинен само на "Ръководител проект". От друга страна, ако погледнем "Програмист 1", той няма нито един подчинен. "Ръководител проект" стои най-високо в йерархията и няма шеф.

По аналогичен начин можем да опишем и ситуацията с останалите участници в проекта. Виждаме как една на пръв поглед малка фигура ни носи много информация.

Терминология, свързана с дърветата

За по-доброто разбиране на тази точка силно препоръчваме на читателя да се опита на всяка стъпка да прави аналогия между тяхното абстрактно значение и това, което използваме в ежедневието.



Нека да опростим начина, по който изобразихме нашата йерархия. Можем да приемем, че тя се състои от точки, свързани с отсечки. За удобство, точките ще номерираме с произволни числа, така че после лесно да можем да говорим за някоя конкретна.

Всяка една точка, ще наричаме **връх**, а всяка една отсечка – **ребро**. Върховете "19", "21" и "14" стоят под върха "7" и са директно свързани с него. Тях ще наричаме **преки наследници (деца)** на "7", а "7" – техен **родител (баща)**. Аналогично "1", "12" и "31" са деца на "19" и "19" е техен родител. Съвсем естествено ще казваме, че "21" е **брат** на "19", тъй като са деца на "7" (обратното също е вярно – "19" е брат на "21"). От гледна точка на "1", "12", "31", "23" и "6", "7" е предшестващ ги в йерархията (в случая е родител на техните родители). Затова "7" ще наречем техен **непряк предшественик (дядо, прародител)**, а тях – негови **непреки наследници**.

Корен е върхът, който няма предшественици. В нашия случай той е "7".

Листа са всички върхове, които нямат наследници. В примера – "1", "12", "31", "21", "23" и "6" са листа.

Вътрешни върхове са всички върхове, различни от корена и листата (т.е. всички върхове, които имат както родител, така и поне един наследник). Такива са "19" и "14".

Път ще наричаме последователност от свързани чрез ребра върхове, в която няма повтарящи се върхове. Например последователността "1", "19", "7" и "21" е път. "1", "19" и "23" не е път, защото "19" и "23" не са свързани помежду си с ребро.

Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Практически този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път ("1", "19", "7" и "21") е три.

Дълбочина на връх ще наричаме дължината на пътя от корена до дадения връх. На примера ни "7" като корен е с дълбочина нула, "19" е с дълбочина едно, а "23" – с дълбочина две.

И така, ето и дефиницията за това какво е дърво:

Дърво (tree) – рекурсивна структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:

- Всеки връх може да има 0 или повече преки наследници (деца).
- Всеки връх има най-много един баща. Съществува точно един специален връх, който няма предшественици – коренът (ако дървото не е празно).
- Всички върхове са достижими от корена, т.е съществува път от корена до всички тях.

Можем да дефинираме дърво и по по-прост начин: всеки единичен връх наричаме дърво и той може да има нула или повече наследници, които също са дървета.

Височина на дърво е максималната от дълбочините на всички върхове. В горния пример височината е 2.

Степен на връх ще наричаме броят на преките наследници (деца) на дадения връх. Степента на "19" и "7" е три, докато тази на "14" е две. Листата са от нулева степен.

Разклоненост на дърво се нарича максималната от степените на всички върхове в дървото. В нашият пример степента на върховете е най-много 3, следователно разклонеността на дървото ни е 3.

Реализация на дърво – пример

Нека сега разгледаме как можем да представяме дърветата като структури от данни в програмирането. Ще реализираме дърво, което съдържа числа във върховете си и всеки връх може да има 0 или повече наследници, които също са дървета (следвайки рекурсивната дефиниция). Всеки връх от дървото е рекурсивно-дефиниран чрез себе си. Един връх от дървото (`TreeNode<T>`) съдържа в себе си списък от наследници, които също са върхове от дървото (`TreeNode<T>`). Нека разгледаме сорс кода:

```
import java.util.ArrayList;

/**
 * Represents a tree data structure.
 * @author Vesko Kolev
 * @param <T> - the type of the values in the tree.
 */
public class Tree<T> {
```

```
/**
 * Represents a tree node.
 * @author Vesko Kolev
 * @param <T> - the type of the values in nodes.
 */
public static class TreeNode<T> {
   // Contains the value of the node
   private T value;

   // Shows whether the current node has parent
   private boolean hasParent;

   // Contains the children of the node
   private ArrayList<TreeNode<T>> children;

   /**
    * Constructs a tree node.
    * @param value - the value of the node.
    */
   public TreeNode(T value) {
      if (value == null) {
         throw new IllegalArgumentException(
           "Cannot insert null value!");
      }
      this.value = value;
      this.children = new ArrayList<TreeNode<T>>();
   }

   /**
    * @return the value of the node.
    */
   public T getValue() {
      return this.value;
   }

   /**
    * Sets the value of the node.
    * @param value - the value to be set.
    */
   public void setValue(T value) {
      this.value = value;
   }

   /**
    * Adds child to the node.
    * @param child - the child to be added.
    */
   public void addChild(TreeNode<T> child) {
```

```
        if (child == null) {
            throw new IllegalArgumentException(
              "Cannot insert null value!");
        }

        if (child.hasParent) {
            throw new IllegalArgumentException(
              "The node already has a parent!");
        }

        child.hasParent = true;
        this.children.add(child);
    }

    /**
     * Gets the child of the node at given index.
     * @param index - the index of the desired child.
     * @return the child on the given position.
     */
    public TreeNode<T> getChild(int index) {
        return this.children.get(index);
    }

    /**
     * @return the number of node's children.
     */
    public int getChildrenCount() {
        return this.children.size();
    }
}

// The root of the tree
private TreeNode<T> root;

/**
 * Constructs the tree.
 * @param value - the value of the node.
 */
public Tree(T value) {
    if (value == null) {
        throw new IllegalArgumentException(
            "Cannot insert null value!");
    }

    this.root = new TreeNode<T>(value);
}

/**
```

```
     * Constructs the tree.
     * @param value - the value of the root node.
     * @param children - the children of the root node.
     */
    public Tree(T value, Tree<T> ...children) {
       this(value);

       for (Tree<T> child : children) {
          this.root.addChild(child.root);
       }
    }

    /**
     * @return the root node or null if the tree is empty.
     */
    public TreeNode<T> getRoot()
    {
       return this.root;
    }

    /**
     * @return the child nodes of the tree.
     */
    public ArrayList<TreeNode<T>> getChildNodes()
    {
       if (this.root != null)
       {
          return this.root.children;
       }
       return new ArrayList<TreeNode<T>>();
    }

    /**
     * Traverses and prints tree in
     * Depth First Search (DFS) manner.
     * @param root - the root of the tree
     * to be traversed.
     * @param spaces - the spaces used for
     * representation of the parent-child relation.
     */
    private void printDFS(TreeNode<T> root, String spaces) {
       if (this.root == null) {
          return;
       }

       System.out.println(spaces + root.getValue());

       TreeNode<T> child = null;
```

```
      for (int i = 0; i < root.getChildrenCount(); i++) {
         child = root.getChild(i);
         printDFS(child, spaces + "   ");
      }
   }

   /**
    * Traverses and prints the tree in
    * Depth First Search (DFS) manner.
    */
   public void printDFS() {
      this.printDFS(this.root, new String());
   }
}

/**
 * Shows a sample usage of the Tree<E> class.
 * @author Vesko Kolev
 */
public class TreeExample {
   public static void main(String[] args) {
      // Create the tree from the sample
      Tree<Integer> tree =
         new Tree<Integer>(7,
               new Tree<Integer>(19,
                     new Tree<Integer>(1),
                     new Tree<Integer>(12),
                     new Tree<Integer>(31)),
               new Tree<Integer>(21),
               new Tree<Integer>(14,
                     new Tree<Integer>(23),
                     new Tree<Integer>(6))
         );

      // Traverse and print the tree using Depth-First-Search
      tree.printDFS();

      // Console output:
      // 7
      //    19
      //       1
      //       12
      //       31
      //    21
      //    14
      //       23
      //       6
   }
```

```
}
```

Как работи нашата имплементация на дърво?

Нека кажем няколко думи за предложения код. В примера имаме клас **Tree<T>**, който е имплементация на самото дървото. В него е дефиниран вътрешен клас – **TreeNode<T>**, който представлява един връх от дървото.

Функционалността, свързана с връх като например създаване на връх, добавяне на наследник на връх, взимане на броя на наследниците и т.н. се реализират на ниво **TreeNode<T>**.

Останалата функционалност (например обхождане на дървото) се реализира на ниво **Tree<T>**. Така функционалността става логически разделена между двата класа, което прави имплементацията по гъвкава.

Причината да разделим на два класа имплементацията е, че някои операции се отнасят за конкретен връх (например добавяне на наследник), докато други се отнасят за цялото дърво (например търсене на връх по неговата стойност). При такова разделяне дървото е клас, който знае кой му е коренът, а всеки връх знае наследниците си. При такава имплементация е възможно да имаме и празно дърво (при **root=null**).

Ето и някои подробности от реализацията на **TreeNode<T>**. Всеки един връх (node) на дървото представлява съвкупност от частно поле **value**, което съдържа стойността му, и списък от наследници **children**. Списъкът на наследниците е от елементи на същия тип. Така всеки връх съдържа списък от референции към неговите преки наследници. Предоставени са също get и set методи за достъп до стойността на върха. Операциите, които могат да се извършват от външен за класа код върху децата, са:

- **addChild(TreeNode<T> child)** - добавя нов наследник.
- **TreeNode<T> getChild(int index)** - връща наследник по зададен индекс.
- **getChildrenCount()** - връща броя на наследници на даден връх.

За да спазим изискването всеки връх в дървото да има точно един родител, сме дефинирали частното поле **hasParent**, което показва дали даденият връх има родител. Тази информация се използва вътрешно в нашия клас и ни трябва в метода **addChild(Tree<E> child)**. В него правим проверка дали кандидат детето няма вече родител. Ако има, се хвърля изключение, показващ, че това е недопустимо.

В класа **Tree<T>** сме предоставили два get метода **TreeNode<T> getRoot()** и **ArrayList<TreeNode<T>> getChildNodes()**, които връщат съответно корена на дървото и неговите преки наследници (деца).

Рекурсивно обхождане на дърво в дълбочина

В класа `Tree<T>` е реализиран е и методът `TraverseDFS()`, който извиква частният метод `DFS(TreeNode<E> root, String spaces)`, който обхожда дървото в дълбочина и извежда на стандартният изход елементите му, така че нагледно да се изобрази дървовидната структура чрез отместване надясно (с добавяне на интервали).

Алгоритъмът за **обхождане в дълбочина** (**Depth-First-Search** или **DFS**) започва от даден връх и се стреми да се спусне колкото се може по-надолу в дървовидната йерархия и когато стигне до връх, от който няма продължение се връща назад към предходния връх. Алгоритъмът можем да опишем схематично по следния начин:

1. Обхождаме текущия връх.
2. Последователно обхождаме рекурсивно всяко едно от поддърветата на текущия връх (обръщаме се рекурсивно към същия метод последователно за всеки един от неговите преки наследници).

Създаване на дърво

За да създаваме по-лесно дървета сме дефинирали специален конструктор, който приема стойност на връх и списък от поддървета за този връх. Така позволяваме подаването на произволен брой аргументи от тип `Tree<E>` (поддървета). При създаването на дървото за нашия пример използваме точно този конструктор и той ни позволява да онагледим структурата му.

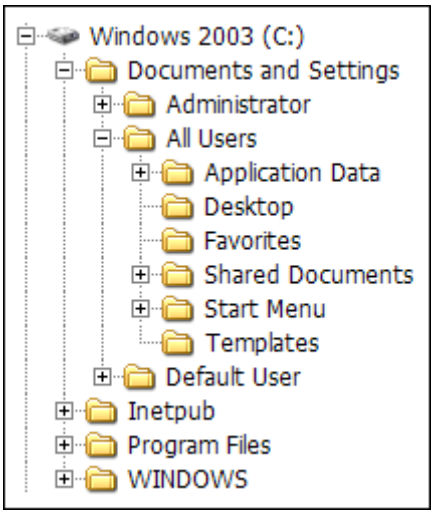
Обхождане на директориите по твърдия диск

Нека сега разгледаме още един пример за дърво – файловата система. Замисляли ли сте се, че директориите върху твърдия ви диск образуват йерархична структура, която е дърво? Можете да се сетите и за много други реални примери, при които се използват дървета.

Нека разгледаме по-подробно файловата система в Windows. Както знаем от нашия всекидневен опит, ние създаваме папки върху твърдия диск, които могат да съдържат от своя страна подпапки или файлове. Подпапките отново може да съдържат подпапки и т. н. до някакво разумно ограничение (максимална дълбочина).

Дървото на файловата система е достъпно чрез стандартни функции от класа `java.io.File`. То не съществува като структура от данни в явен вид, но съществува начин да извличаме за всяка директория файловете и директориите в нея и следователно можем да го обходим чрез стандартен алгоритъм за обхождане на дървета.

Ето как изглежда типичното дърво на директориите в Windows:



Рекурсивно обхождане на директориите в дълбочина

Следващият пример показва как да обходим рекурсивно (в дълбочина, по алгоритъма Depth-First-Search) дървовидната структура на дадена папка и да изведем на стандартния изход и нейното съдържание:

DirectoryTraverserDFS.java

```
import java.io.File;

/**
 * Sample class, which traverses recursively given directory
 * based on the Depth-First-Search (DFS) algorithm.
 *
 * @author Vesko Kolev
 */
public class DirectoryTraverserDFS {
   /**
    * Traverses and prints given directory recursively.
    * @param dir - the directory to be traversed.
    * @param spaces - the spaces used for representation
    *        of the parent-child relation.
    */
   private static void traverseDir(File dir, String spaces) {

      // If the current element is a directory,
      // we get all it subdirectories and files
      if (dir.isDirectory()) {
         System.out.println(spaces + dir.getAbsolutePath());
         String[] children = dir.list();
```

```
        // For each child go and visit its subtree
        for (String child : children) {
          traverseDir(new File(dir, child), spaces + "   ");
        }
      }
    }

    /**
     * Traverses and prints given directory recursively.
     * @param directoryPath - the path to the directory which
     *        should be traversed.
     */
    public static void traverseDir(String directoryPath) {
      traverseDir(new File(directoryPath), new String());
    }

    public static void main(String[] args) {
      traverseDir("C:\\");
    }
}
```

Както се вижда от примера, рекурсивното обхождане на съдържанието на директория по нищо не се различава от обхождането на нашето дърво. В случая, файловете представляват листа, началната директория – корен, а останалите директории – вътрешни върхове. Всъщност, ако една директория е празна, тогава тя също се явява листо, защото няма наследници.

Ето как изглежда резултатът от обхождането (със съкращения):

```
C:\
  C:\Config.Msi
  C:\Documents and Settings
    C:\Documents and Settings\Administrator
      C:\Documents and Settings\Administrator\.ARIS70
      C:\Documents and Settings\Administrator\.jindent
      C:\Documents and Settings\Administrator\.nbi
        C:\Documents and Settings\Administrator\.nbi\downloads
        C:\Documents and Settings\Administrator\.nbi\log
        C:\Documents and Settings\Administrator\.nbi\cache
        C:\Documents and Settings\Administrator\.nbi\tmp
        C:\Documents and Settings\Administrator\.nbi\wd
      C:\Documents and Settings\Administrator\.netbeans
        C:\Documents and Settings\Administrator\.netbeans\6.0
...
```

Обхождане на директориите в ширина

Нека сега разгледаме още един начин да обхождаме дървета. **Обхождането в ширина** (**Breath-First-Search** или **BFS**) е алгоритъм за обхождане на дървовидни структури от данни, при който първо се посещава началният връх, след това неговите преки съседи, след тях преките съседи на съседите и т.н. Този процес **метод на вълната**, защото прилича на вълните, образувани от камък, хвърлен в езеро.

Алгоритъмът за обхождане на дърво в ширина по метода на вълната можем да опишем схематично по следния начин:

1. Записваме в опашката `Q` началния връх.
2. Докато `Q` не е празна повтаряме следните две стъпки:
   - Изваждаме от `Q` поредния връх `v` и го отпечатваме.
   - Добавяме всички наследници на `v` в опашката.

Алгоритъмът BFS е изключително прост и има свойството да обхожда първо най-близките до началния връх върхове, след тях по-далечните и т.н. и най-накрая – най-далечните върхове. С времето ще се убедите, че BFS алгоритъмът има широко приложение при решаването на много задачи, като например при търсене на най-кратък път в лабиринт.

Нека сега приложим BFS алгоритъма за отпечатване на всички директории от файловата система:

DirectoryTraverserBFS.java

```
import java.io.File;
import java.util.LinkedList;
import java.util.Queue;

/**
 * Sample class, which traverses given directory
 * based on the Breath-First-Search (BFS) algorithm.
 * @author Svetlin Nakov
 */
public class DirectoryTraverserBFS {
   /**
    * Traverses and prints given directory with BFS.
    * @param startDir - the path to the directory which
    *        should be traversed.
    */
   public static void traverseDir(String startDirectory) {
      Queue<File> visitedDirsQueue = new LinkedList<File>();
      visitedDirsQueue.add(new File(startDirectory));
      while (visitedDirsQueue.size() > 0) {
         File currentDir = visitedDirsQueue.remove();
```

```
        System.out.println(currentDir.getAbsolutePath());
        File[] children = currentDir.listFiles();
        if (children != null) {
           for (File child : children) {
              if (child.isDirectory()) {
                 visitedDirsQueue.add(child);
              }
           }
        }
     }
   }

   public static void main(String[] args) {
      traverseDir("C:\\");
   }
}
```

Ако стартираме програмата, ще се убедим, че обхождането в ширина първо открива най-близките директории до корена (дълбочина 1), след тях всички директории на дълбочина 2, след това директориите на дълбочина 3 и т.н. Ето примерен изход от програмата:

```
C:\
C:\Config.Msi
C:\Documents and Settings
C:\Inetpub
C:\Program Files
C:\RECYCLER
C:\System Volume Information
C:\WINDOWS
C:\wmpub
C:\Documents and Settings\Administrator
C:\Documents and Settings\All Users
C:\Documents and Settings\Default User
...
```

Двоични дървета

В предишната точка от темата разгледахме обобщената структура дърво. Сега ще преминем към един неин полезен частен случай, който се оказва изключително важен за практиката – **двоично дърво**. Важно е да отбележим, че термините, които дефинирахме до момента, важат с пълна сила и при този вид дърво. Въпреки това по-долу ще дадем и някои допълнителни, специфични за дадената структура определения.

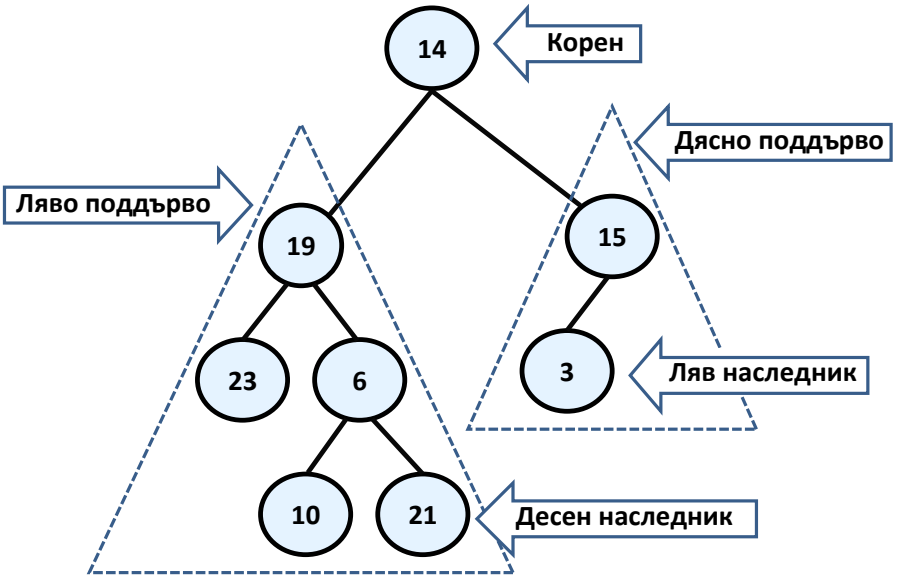
Двоично дърво (binary tree) – дърво, в което всеки връх е от степен не надвишаваща две т.е. дърво с разклоненост две. Тъй като преките наследници (деца) на всеки връх са най-много два, то е прието да се

въвежда наредба между тях, като единият се нарича **ляв наследник**, а другият – **десен наследник**. Те от своя страна са корени съответно на **лявото поддърво** и на **дясното поддърво** на техния родител.

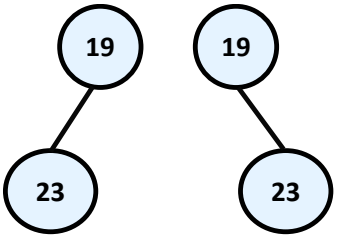
Двоично дърво – пример

Ето и едно примерно двоично дърво, което ще използваме за изложението по-нататък. В този пример отново въвеждаме номерация на върховете, която е абсолютно произволна и която ще използваме, за да може по-лесно да говорим за всеки връх.

На примера са изобразени съответно коренът на дървото "14", пример за ляво поддърво (с корен "19") и дясно поддърво (с корен "15"), както и ляв и десен наследник – съответно "3" и "21".

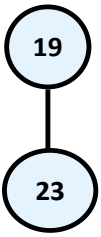


Следва да отбележим обаче, че двоичните дървета имат едно много сериозно различие в дефиницията си, за разлика от тази на обикновеното дърво – наредеността на наследниците на всеки връх. Следващият пример ясно показва това различие:



На схемата са изобразени две абсолютно различни **двоични дървета** – в единия случай коренът е "19" и има **ляв наследник** "23", а в другия имаме двоично дърво с корен отново "19", но с "23" за **десен наследник**. Ако

разгледаме обаче двете структури като **обикновени дървета**, те ще са абсолютно еднакви и неразличими. Затова такова **дърво** бихме изобразили по следния начин:



Запомнете! Въпреки, че разглеждаме двоичните дървета като подмножество на структурата дърво, трябва да се отбележи, че условието за наредености на наследниците ги прави до голяма степен различни като структури.

Обхождане на двоично дърво

Обхождането на дърво по принцип е една класическа и често срещана задача. В случая на двоичните дървета има няколко основни начина за обхождане:

- **ЛДК (Ляво-Корен-Дясно/Pre-order)** – обхождането става като първо се обходи лявото поддърво, след това коренът и накрая дясното поддърво. В нашият пример последователността, която се получава при обхождането е: "23", "19", "10", "6", "21", "14", "3", "15".
- **КЛД (Корен-Ляво-Дясно/In-order)** – в този случай първо се обхожда коренът на дървото, после лявото поддърво и накрая дясното. Ето и как изглежда резултатът от този вид обхождане: "14", "19", "23", "6", "10", "21", "15", "3".
- **ЛДК (Ляво-Дясно-Корен/Post-order)** – тук по аналогичен на горните два примера начин, обхождаме първо лявото поддърво, после дясното и накрая коренът. Резултатът след обхождането е "23", "10", "21", "6", "19", "3", "15", "14".

Обхождане на двоично дърво с рекурсия – пример

В следващия пример ще покажем примерна реализация на двоично дърво, което ще обходим по схемата ЛДК:

```
/**
 * Represents a binary tree structure.
 * @author Vesko Kolev
 */
public class BinaryTree<T> {
```

```
/**
 * Represents a binary tree node.
 * @author Vesko Kolev
 * @param <T> - the type of the values in nodes.
 */
public static class BinaryTreeNode<T>
{
   // Contains the value of the node
   private T value;

   // Shows whether the current node has parent
   private boolean hasParent;

   // Contains the left child of the node
   private BinaryTreeNode<T> leftChild;

   // Contains the right child of the node
   private BinaryTreeNode<T> rightChild;

   /**
    * Constructs a binary tree node.
    * @param value - the value of the node.
    * @param leftChild - the left child of the node.
    * @param rightChild - the right child of the node.
    */
   public BinaryTreeNode(T value,
        BinaryTreeNode<T> leftChild,
        BinaryTreeNode<T> rightChild)
   {
      if (value == null) {
         throw new IllegalArgumentException(
            "Cannot insert null value!");
      }

      this.value = value;
      this.leftChild = leftChild;
      this.rightChild = rightChild;
   }

   /**
    * Constructs a binary tree node with no children.
    * @param value - the value of the node.
    */
   public BinaryTreeNode(T value)
   {
      this(value, null, null);
   }
```

```
    /**
     * @return the value of the node.
     */
    public T getValue() {
       return this.value;
    }

    /**
     * Sets the value of the node.
     * @param value - the value to be set.
     */
    public void setValue(T value) {
       this.value = value;
    }

    /**
     * @return the left child or null if it does not exists.
     */
    public BinaryTreeNode<T> getLeftChild() {
       return this.leftChild;
    }

    /**
     * Sets the left child.
     * @param value - the new left child to be set.
     */
    public void setLeftChild(BinaryTreeNode<T> value) {
       if (value == null || value.hasParent) {
          throw new IllegalArgumentException();
       }

       value.hasParent = true;
       this.leftChild = value;
    }

    /**
     * @return the right child or null if it does not exists.
     */
    public BinaryTreeNode<T> getRightChild() {
       return this.rightChild;
    }

    /**
     * Sets the right child.
     * @param value - the new right child to be set.
     */
    public void setRightChild(BinaryTreeNode<T> value) {
       if (value == null || value.hasParent) {
```

```
            throw new IllegalArgumentException();
        }

        value.hasParent = true;
        this.rightChild = value;
    }
}

// The root of the tree
private BinaryTreeNode<T> root;

/**
 * Constructs the tree.
 * @param value - the value of the node.
 * @param children - the children of the node.
 */
public BinaryTree(T value, BinaryTree<T> leftChild,
        BinaryTree<T> rightChild) {
    if (value == null) {
        throw new IllegalArgumentException();
    }

    BinaryTreeNode<T> leftChildNode =
        leftChild != null ? leftChild.root : null;
    BinaryTreeNode<T> rightChildNode =
        rightChild != null ? rightChild.root : null;
    this.root = new BinaryTreeNode<T>(
        value, leftChildNode, rightChildNode);
}

/**
 * Constructs the tree.
 * @param value - the value of the node.
 */
public BinaryTree(T value) {
    this(value, null, null);
}

/**
 * @return the root of the tree.
 */
public BinaryTreeNode<T> getRoot()
{
    return this.root;
}

/**
 * @return the left child of the root.
```

```
     */
    public BinaryTreeNode<T> getLeftChildNode()
    {
       if (this.root != null)
       {
          return this.root.getLeftChild();
       }

       return null;
    }

    /**
     * @return the right child of the root.
     */
    public BinaryTreeNode<T> getRightChildNode()
    {
       if (this.root != null)
       {
          return this.root.getRightChild();
       }

       return null;
    }

     /**
      * Traverses binary tree in pre-order manner.
      * @param root - the binary tree to be traversed.
      */
     private void printPreOrder(BinaryTreeNode<T> root) {
      if (root == null) {
         return;
      }

      // 1. Visit the left child.
      printPreOrder(root.getLeftChild());

      // 2. Visit the root of this subtree.
      System.out.print(root.getValue() + " ");

      // 3. Visit the right child.
      printPreOrder(root.getRightChild());
     }

     /**
      * Traverses and prints the binary
      * tree in pre-order manner.
      */
     public void printPreOrder() {
```

```
      printPreOrder(this.root);
      System.out.println();
    }
}

/**
 * Shows how the BinaryTree class can be used.
 * @author Vesko Kolev
 */

public class BinaryTreeExample {
   public static void main(String[] args) {
      // Create the binary tree from the sample.
      BinaryTree<Integer> binaryTree =
         new BinaryTree<Integer>(14,
               new BinaryTree<Integer>(19,
                     new BinaryTree<Integer> (23),
                     new BinaryTree<Integer> (6,
                           new BinaryTree<Integer>(10),
                           new BinaryTree<Integer>(21))),
               new BinaryTree<Integer>(15,
                     new BinaryTree<Integer>(3),
                     null));

      // Traverse and print the tree in pre-order manner.
      binaryTree.printPreOrder();

      // Console output:
      // 23 19 10 6 21 14 3 15
   }
}
```

Как работи примерът?

Тази примерна имплементация на двоично дърво не се различава съществено от реализацията, която показахме в случая на обикновено дърво. Отново имаме отделни класове за представяне на двоично дърво и на връх в такова – **BinaryTree<T>** и **BinaryTreeNode<T>**. Във вътрешния клас **BinaryTreeNode<T>** имаме частни полета **value** и **hasParent**. Както и преди първото съдържа стойността на върха, а второто показва дали върха има родител. При добавяне на ляв или десен наследник (ляво/дясно дете) на даден връх, се прави проверка дали имат вече родител и ако имат, се хвърля изключение, аналогично на реализацията ни на дърво.

За разлика от реализацията на обикновеното дърво, сега вместо списък на децата, всеки връх съдържа по едно частно поле за ляв и десен наследник. За всеки от тях сме дефинирали публични get и set методи, за да могат да се достъпват от външен за класа код.

В `BinaryTree<T>` са реализирани три `get` метода, които връщат съответно корена на дървото, левия му наследник и десния му наследник. Методът `traversePreOrder()` извиква вътрешно метода `preOrder(BinaryTreeNode< T> root)`. Вторият метод от своя страна обхожда подаденото му дърво по схемата ляво-корен-дясно (ЛКД). Това става по следния тристъпков алгоритъм:

1. Рекурсивно извикване на метода за обхождане за лявото поддърво на дадения връх.
2. Обхождане на самия връх.
3. Рекурсивно извикване на метода за обхождане на дясното поддърво.

Силно препоръчваме на читателя да се опита (като едно добро упражнение) да модифицира предложения алгоритъм и код самостоятелно, така че да реализира другите два основни типа обхождане.

Наредени двоични дървета за претърсване

До момента видяхме как можем да построим обикновено дърво и двоично дърво. Тези структури сами по себе си са доста обобщени и трудно, в такъв суров вид, могат да ни свършат някаква по-сериозна работа. На практика в информатиката се прилагат някои техни разновидности, в които са дефинирани съвкупност от строги правила (алгоритми) за различни операции с тях и с техните елементи. Всяка една от тези разновидности носи със себе си специфични свойства, които са полезни в различни ситуации.

Като примери за такива полезни свойства могат да се дадат бързо търсене на елемент по зададена стойност (червено-черно дърво); нареденост (сортираност) на елементите в дървото; възможност да се организира големи количества информация на някакъв файлов носител, така че търсенето на елемент в него да става бързо с възможно най-малко стъпки (B-дърво) както и много други.

В тази секция ще разгледаме един по-специфичен клас двоични дървета – **наредените**. Те използват едно често срещано при двоичните дървета свойство на върховете, а именно съществуването на **уникален идентификационен ключ** във всеки един от тях. Този ключ не се среща никъде другаде в рамките на даденото дърво. Наредените двоични дървета позволяват бързо (в общия случай с приблизително log(n) на брой стъпки) търсене, добавяне и изтриване на елемент, тъй като поддържат елементите си индиректно в сортиран вид.

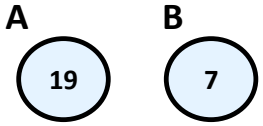
Сравнимост между обекти

Преди да продължим, ще въведем следната дефиниция, от която ще имаме нужда в по-нататъшното изложение.

Сравнимост – два обекта A и B наричаме сравними, ако е изпълнена **точно една** от следните три зависимости между тях:
- "A е по-малко от B"
- "A е по-голямо от B"
- "A е равно на B"

Аналогично два ключа A и B ще наричаме сравними, ако е изпълнена точно една от следните три възможности: A < B, A > B или A = B.

Върховете на едно дърво могат да съдържат най-различни полета. В по-нататъшното разсъждение ние ще се интересуваме само от техните уникални ключове, които ще искаме да са сравними. Да покажем един пример. Нека са дадени два конкретни върха A и B:



В примера ключът на A и B са съответно целите числа 19 и 7. Както знаем от математиката, целите числа (за разлика от комплексните например) са **сравними**, което според гореизложените разсъждения ни дава правото да ги използваме като ключове. Затова за върховете A и B можем да кажем, че "A е по-голямо от B" тъй като "19 е по-голямо от 7".

Забележете! Този път числата изобразени във върховете са техни уникални идентификационни ключове, а не както досега произволни числа.

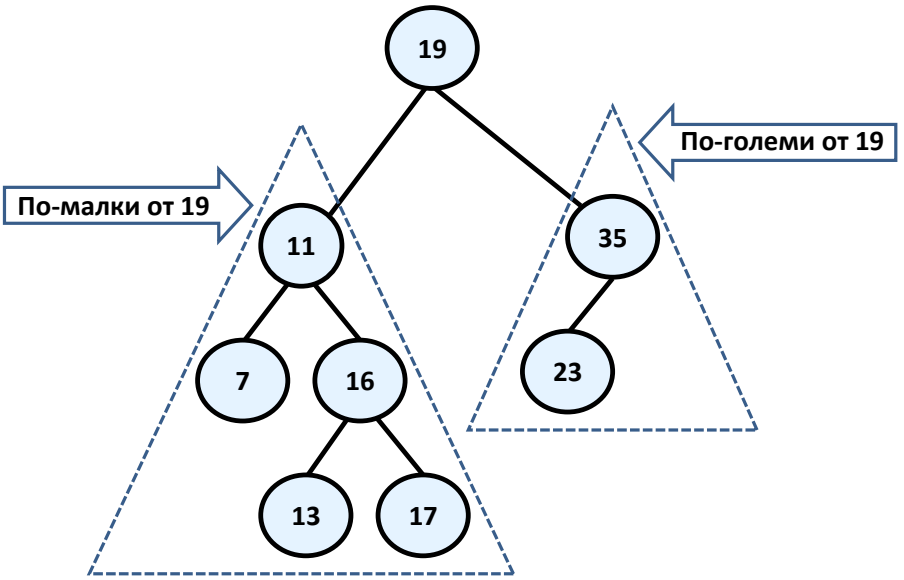
Стигаме и до дефиницията за **наредено двоично дърво за търсене:**

Наредено двоично дърво (дърво за търсене, binary search tree) е двоично дърво, в което всеки два от ключовете са сравними и което е организирано, така че за всеки връх да е изпълнено:

- Всички ключове в **лявото** му поддърво са **по-малки** от неговия ключ.
- Всички ключове в **дясното** му поддърво са **по-големи** от неговия ключ.

Свойства на наредените двоични дървета за претърсване

На фигурата е изобразен пример за наредено двоично дърво за претърсване. Ще използваме този пример, за да дадем някой важни свойства на наредеността на двоично дърво:



По дефиниция имаме, че лявото поддърво на всеки един от върховете се състои само от елементи, които са по-малки от него, докато в дясното поддърво има само по-големи елементи. Това означава, че ако искаме да намерим даден елемент тръгвайки от корена, то или сме го намерили или трябва да го търсим съответно в лявото или дясното му поддърво, с което ще спестим излишни сравнения. Например, ако търсим в нашето дърво 23, то няма смисъл да го търсим в лявото поддърво на 19, защото 23 със сигурност не е там (23 е по-голямо от 19 следователно евентуално е в дясното поддърво). Това ни спестява 5 излишни сравнения с всеки един от елементите от лявото поддърво, които, ако използваме свързан списък например, ще трябва да извършим.

От наредеността на елементите следва, че **най-малкият** елемент в дървото е най-левият наследник на корена, ако има такъв, или самият корен, ако той няма ляв наследник. По абсолютно същия начин **най-големият** елемент в дървото е най-десният наследник на корена, а ако няма такъв – самият корен. В нашия пример това са минималният елемент 7 и максималният – 35. Полезно и директно следващо свойство от това е, че всеки един елемент от лявото поддърво на даден връх е по-малък от всеки друг, който е в дясното поддърво на същия връх.

Наредени двоични дървета за търсене – пример

Следващият пример показва реализация на двоично дърво за търсене. Целта ни ще бъде да предложим методи за добавяне, търсене и изтриване на елемент в дървото. За всяка една от тези операции ще дадем подробно обяснение как точно се извършва.

Наредени двоични дървета: реализация на върховете

Както и преди, сега ще дефинираме вътрешен клас, който да опише структурата на един връх. По този начин ясно ще разграничим и капсулираме структурата на един връх като същност, която дървото ни ще съдържа в себе си. Този отделен клас сме дефинирали като частен и е видим само в класа на наредената ни дърво. Ето и неговата дефиниция:

```
...
private static class BinaryTreeNode<T extends Comparable<T>>
        implements Comparable<BinaryTreeNode<T>> {
   // Contains the value of the node
   T value;

   // Contains the parent of the node
   BinaryTreeNode<T> parent;

   // Contains the left child of the node
   BinaryTreeNode<T> leftChild;

   // Contains the right child of the node
   BinaryTreeNode<T> rightChild;
   /**
    * Constructs the tree node.
    * @param value – the new value.
    */
   public BinaryTreeNode(T value) {
      this.value = value;
      this.parent = null;
      this.leftChild = null;
      this.rightChild = null;
   }

   @Override
   public String toString() {
      return this.value.toString();
   }

   @Override
   public int hashCode() {
      return this.value.hashCode();
   }

   @Override
   public boolean equals(Object obj) {
      BinaryTreeNode<T> other = (BinaryTreeNode<T>)obj;
      return this.compareTo(other) == 0;
   }
```

```
   public int compareTo(BinaryTreeNode<T> other) {
      return this.value.compareTo(other.value);
   }
}
...
```

Да разгледаме предложения код. Още в името на структурата, която разглеждаме – "наредено дърво за търсене", ние говорим за наредба, а такава можем да постигнем **само** ако имаме **сравнимост** между елементите в дървото.

Сравнимост между обекти в Java

Какво означава понятието "сравнимост между обекти" за нас като програмисти? Това означава, че трябва да задължим по някакъв начин всички, които използват нашата структура от данни, да я създават подавайки и **тип, който е сравним**. На Java изречението "тип, който е сравним" би "звучало" така:

```
T extends Comparable<T>
```

Интерфейсът **`Comparable<T>`**, намиращ се в пакета **`java.lang`**, се състои само от един метод **`int compareTo(T obj)`**, който връща отрицателно цяло число, нула или положително цяло число съответно, ако текущият обект е по-малък, равен или по-голям от този, който е подаден на метода. Дефиницията му изглежда по приблизително следния начин:

```
public interface Comparable<T> {
   /**
    * Compares this object with the specified object for order.
    * @param obj - the Object to be compared
    * @return a negative integer, zero, or a positive integer as
    * this object is less than, equal to, or greater than the
    * specified object.
    */
   int compareTo(T obj);
}
```

Имплементирането на този интерфейс от даден клас ни гарантира, че неговите инстанции са сравними.

От друга страна на нас ни е необходимо и самите върхове описани чрез класа **`BinaryTreeNode`** също да бъдат сравними помежду си. Затова той също имплементира **`Comparable<T>`**. Както се вижда от кода, имплементацията на **`Comparable<T>`** на класа **`BinaryTreeNode`** вътрешно извиква тази на типа **`T`**.

В кода също сме припокрили и методите **equals(Object obj)** и **hashCode()**. Добра (задължителна) практика е тези два метода да са съгласувани в поведението си т.е. когато два обекта са еднакви, хеш-кодът им да е еднакъв. Както ще видим в главата за хеш-таблици, обратното въобще не е задължително. Аналогично очакваното поведение на **equals(Object obj)** е да връща истина, точно когато и **compareTo(T obj)** връща 0.

Задължително синхронизирайте работата на методите equals(Object obj), compareTo(T obj) и hashCode(). Това е тяхното очаквано поведение и ще ви спести много трудно откриваеми проблеми!

До тук разгледахме методите, предложени от нашият клас. Сега да видим какви полета ни предоставя. Те са съответно за **value** (ключът) от тип **T** родител – **parent**, ляв и десен наследник – съответно **leftChild** и **rightChild**. Последните три са от типа на дефиниращия ги клас, а именно **BinaryTreeNode**.

Наредени двоични дървета: реализация на основния клас

Преминаваме към реализацията на класа, описващ самото наредено двоично дърво. Дървото само по себе си като структура се състои от един корен от тип **BinaryTreeNode**, който вътрешно съдържа наследниците си – съответно ляв и десен, те вътрешно техните наследници и така рекурсивно надолу докато се стигне до листата. Друго важно за отбелязване нещо е дефиницията **BinarySearchTree<T extends Comparable<T>>**. Това ограничение на типа **T** се налага заради изискването на вътрешния ни клас, който работи само с типове, имплементиращи **Comparable<T>**.

```
public class BinarySearchTree<T extends Comparable<T>> {

   /**
    * Represents a binary tree node.
    * @author Vesko Kolev
    * @param <T>
    */
   private static class BinaryTreeNode<T extends Comparable<T>>
           implements Comparable<BinaryTreeNode<T>> {
      //...
      //... The implementation from above goes here!!! ...
      //...
   }

   /**
    * The root of the tree.
    */
   private BinaryTreeNode<T> root;
```

```
   /**
    * Constructs the tree.
    */
   public BinarySearchTree() {
      this.root = null;
   }

   //...
   //... The operation implementation goes here!!! ...
   //...

}
```

Както споменахме по-горе, ще разгледаме следните операции:

- добавяне на елемент;
- търсене на елемент;
- изтриване на елемент.

Добавяне на елемент в подредено двоично дърво

След добавяне на нов елемент, дървото трябва да запази своята нареденост. Алгоритъмът е следният: ако дървото е празно, то добавяме новия елемент като корен. В противен случай:

- Ако елементът е по-малък от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в лявото поддърво.
- Ако елементът е по-голям от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в дясното поддърво.
- Ако елементът е равен на корена, то не правим нищо и излизаме от рекурсията.

Ясно се вижда как алгоритъмът за включване на връх изрично се съобразява с правилото елементите в лявото поддърво да са по-малки от корена на дървото и елементите от дясното поддърво да са по-големи от корена на дървото. Ето и примерна имплементация на този метод. Забележете, че при включването се поддържа референция към родителя, защото родителят също трябва да бъде променен.

```
/**
 * Inserts new value in the binary search tree.
 * @param value - the value to be inserted.
 */
public void insert(T value) {
   if (value == null) {
      throw new IllegalArgumentException();
```

```
    }

    this.root = insert(value, null, root);
}

/**
 * Inserts node in the binary search tree by given value.
 * @param value - the new value.
 * @param parentNode - the parent of the new node.
 * @param node - current node.
 * @return the inserted node
 */
private BinaryTreeNode<T> insert(T value,
      BinaryTreeNode<T> parentNode,
      BinaryTreeNode<T> node) {
    if (node == null) {
      node = new BinaryTreeNode<T>(value);
      node.parent = parentNode;
    } else {
      int compareTo = value.compareTo(node.value);
      if (compareTo < 0) {
        node.leftChild =
          insert(value, node, node.leftChild);
      } else if (compareTo > 0) {
        node.rightChild =
          insert(value, node, node.rightChild);
      }
    }

    return node;
}
```

Търсене на елемент в подредено двоично дърво

Търсенето е операция, която е още по-интуитивна. В примерния код сме показали как може търсенето да се извърши без рекурсия, чрез итерация. Алгоритъмът започва с елемент **node**, сочещ корена. След това се прави следното:

- Ако елементът е равен на **node**, то сме намерили търсения елемент и го връщаме.
- Ако елементът е по-малък от **node**, то присвояваме на **node** левия му наследник т.е. продължаваме търсенето в лявото поддърво.
- Ако елементът е по-голям от **node**, то присвояваме на **node** десния му наследник т.е. продължаваме търсенето в дясното поддърво.

Следва примерен код:

```
/**
 * Finds a given value in the tree and returns the node
 * which contains it if such exsists.
 * @param value - the value to be found.
 * @return the found node or null if not found.
 */
private BinaryTreeNode<T> find(T value) {
   BinaryTreeNode<T> node = this.root;

   while (node != null) {
      int compareTo = value.compareTo(node.value);
      if (compareTo < 0) {
         node = node.leftChild;
      } else if (compareTo > 0) {
         node = node.rightChild;
      } else {
         break;
      }
   }

   return node;
}
```

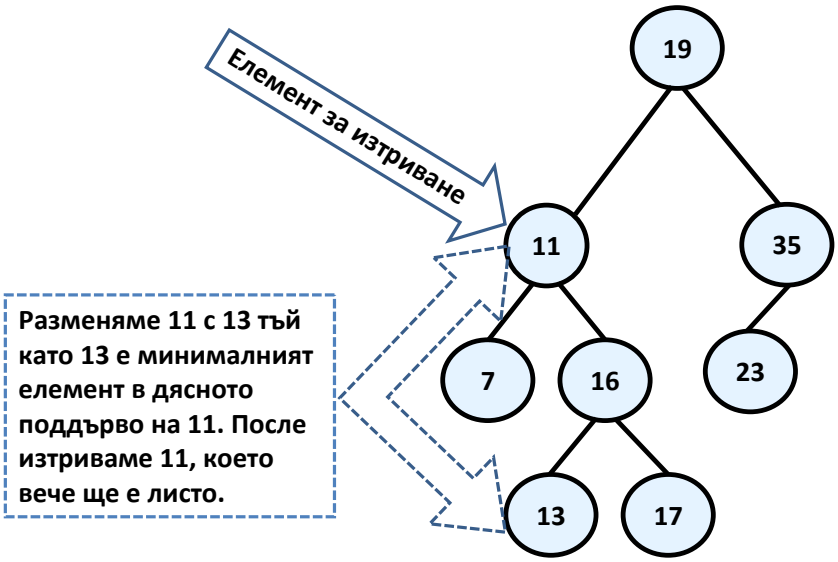
Изтриване на елемент от подредено двоично дърво

Изтриването е най-сложната операция от трите основни. След нея дървото трябва да запази своята наредеността. Първата стъпка преди да изтрием елемент от дървото е да го намерим. Вече знаем как става това. След това се прави следното:

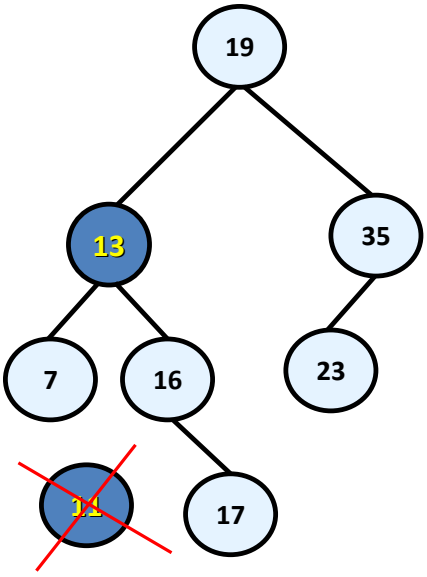
- Ако върхът е листо – насочваме референцията на родителя му към **null**. Ако елементът няма родител следва, че той е корен и просто го изтриваме.
- Ако върхът има само едно поддърво – ляво или дясно, то той се замества с корена на това поддърво.
- Ако върхът има две поддървета. Тогава намираме най-малкият връх в дясното му поддърво и го разменяме с него. След тази размяна върхът ще има вече най-много едно поддърво и го изтриваме по някое от горните две правила. Тук трябва да отбележим, че може да се направи аналогична размяна, само че взимаме лявото поддърво и най-големият елемент от него.

Оставяме на читателя като леко упражнение да провери коректността на всяка от тези три стъпки.

Нека даден един пример за изтриване. Ще използваме отново нашето наредено дърво, което показахме в началото на тази точка. Да изтрием например елемента с ключ 11.



Той има две поддървета и съгласно нашият алгоритъм трябва да бъде разменен с най-малкият елемент от дясното поддърво, т.е. с 13. След като извършим размяната вече можем спокойно да изтрием 11, който е листо. Ето крайният резултат:



Предлагаме следния примерен код, който реализира описания алгоритъм:

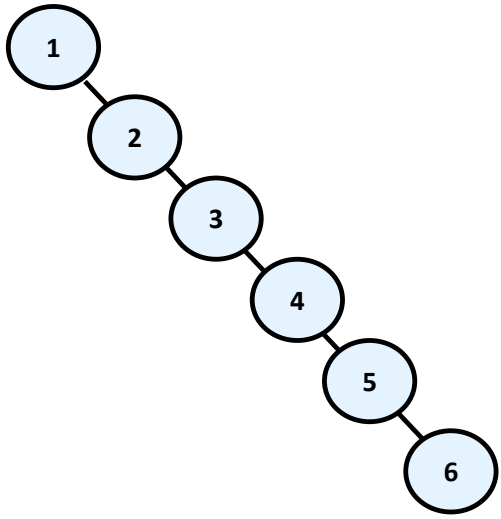
```
/**
 * Removes an element from the tree if exists.
 * @param value - the value to be deleted.
 */
```

```
public void remove(T value) {
   BinaryTreeNode<T> nodeToDelete = find(value);
   if (nodeToDelete == null) {
      return;
   }

   remove(nodeToDelete);
}

public void remove(BinaryTreeNode<T> node) {
   // Case 3: If the node has two children.
   // Note that if we get here at the end
   // the node will be with at most one child.
   if (node.leftChild != null && node.rightChild != null) {
      BinaryTreeNode<T> replacement = node.rightChild;
      while (replacement.leftChild != null) {
         replacement = replacement.leftChild;
      }
      node.value = replacement.value;
      node = replacement;
   }

   // Case 1 and 2: If the node has at most one child.
   BinaryTreeNode<T> theChild = node.leftChild != null ?
         node.leftChild : node.rightChild;

   // If the element to be deleted has one child.
    if (theChild != null) {
         theChild.parent = node.parent;

         // Handle the case when the element is the root.
         if (node.parent == null) {
            root = theChild;
         }
         else {
            // Replace the element with its child subtree.
            if (node.parent.leftChild == node) {
               node.parent.leftChild = theChild;
            }
            else {
               node.parent.rightChild = theChild;
            }
         }
    }
    else {
     // Handle the case when the element is the root.
     if (node.parent == null) {
        root = null;
```

```
        }
        else {
        // Remove the element. It is a leaf.
        if (node.parent.leftChild == node) {
              node.parent.leftChild = null;
          }
          else {
              node.parent.rightChild = null;
          }
        }
    }
}
```

Балансирани дървета

Както видяхме по-горе, наредените двоични дървета представляват една много удобна структура за търсене. Така дефинирани операциите за създаване и изтриване на дървото имат един скрит недостатък. Какво би станало ако в дървото включим последователно елементите 1, 2, 3, 4, 5, 6? Ще се получи следното дърво:



В този случай двоичното дърво се е изродило в свързан списък. От там и търсенето в това дърво ще е доста по-бавно (с N на брой стъпки, а не с log(N)), тъй като, за да проверим дали даден елемент е вътре, в най-лошият случай ще трябва да преминем през всички елементи.

Ще споменем накратко за съществуването на структури от данни, които в общия случай запазват логаритмичното поведение на операциите добавяне, търсене и изтриване на елемент. Преди да кажем как се постига това, ще въведем следните две дефиниции:

Балансирано двоично дърво – двоично дърво, в което никое листо не е на "много по-голяма" дълбочина от всяко друго листо. Дефиницията на "много по-голяма" зависи от конкретната балансираща схема.

Идеално балансирано двоично дърво – двоично дърво, в което разликата в ***броя на върховете на лявото и дясното поддърво*** на всеки от върховете е най-много единица.

Без да навлизаме в детайли ще споменем, че ако дадено двоично дърво е балансирано, дори и да не е идеално балансирано, то операциите за добавяне, търсене и изтриване на елемент в него са с логаритмична сложност и дори и в най-лошия случай. За да се избегне дисбаланса на дървото за претърсване, се прилагат операции, които пренареждат част от елементите на дървото при добавяне или при премахване на елемент от него. Тези операции най-често се наричат **ротации**. Конкретният вид на ротациите, се уточнява допълнително и зависи реализацията от конкретната структура от данни. Като примери за такива структури, можем да дадем **червено-черно** дърво, **AVL**-дърво, **AA**-дърво, **Splay**-дърво и др.

За по-детайлно разглеждане на тези и други структури препоръчваме на читателя да потърси в строго специализираната литература за алгоритми и структури от данни.

Класът TreeSet<T> в Java

След като вече се запознахме с наредените двоични дървета и с това какво е предимството те да са балансирани, идва момента да покажем и какво Java има за нас по този въпрос. Може би всеки от вас тайно се е надявал, че никога няма да му се налага да имплементира балансирано наредено двоично дърво за търсене, защото изглежда доста сложно. Това най-вероятно наистина е така.

До момента разгледахме какво представляват балансираните дървета, за да добиете представа за тях. Когато ви се наложи да ги ползвате, винаги можете да разчитате да ги вземете от някъде наготово. В стандартните библиотеки на Java има готови имплементации на балансирани дървета, а освен това по Интернет можете да намерите и много външни библиотеки, като примерно Apache Commons Collections и JGL.

В Java Collection Framework се поддържа класът `TreeSet<T>`, който вътрешно представлява имплементация на червено-черно дърво. Това, както вече знаем, означава, че добавянето, търсенето и изтриването на елементи в дървото ще се извърши с логаритмична сложност (т.е. ако имаме 1 000 000 елемента операцията ще бъде извършена за около 20 стъпки). Методите са съответно с имена `add()`, `contains()` и `remove()`.

Важно е да се отбележи, че итераторът на `TreeSet<T>` ще връща елементите на дървото един по един в нарастващ ред, като се започне от най-малкия. Това се дължи на вътрешната наредба на елементите. Следва да отбележим, че итераторите са класове, които се използват за обхождане на

дадена колекция. Има случаи, в които се имплементират повече от един итератори, за да могат да бъдат постигнати няколко различни обхождания на дадената структура.

Класът TreeSet<T> – пример

Ето прост пример, който показва, че в TreeSet<T> можем да добавяме и изтриваме, а при обхождане получаваме елементите в нарастващ ред:

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
treeSet.add(5);
treeSet.add(8);
treeSet.add(1);
treeSet.add(6);
treeSet.add(3);
treeSet.remove(6);
for (int i : treeSet) {
   System.out.printf(" %d", i);
}
// Result:  1 3 5 8
```

Повече информация за класа **TreeSet<T>** можете да намерите в секцията "Множества" на главата "Речници, хеш-таблици и множества".

Графи

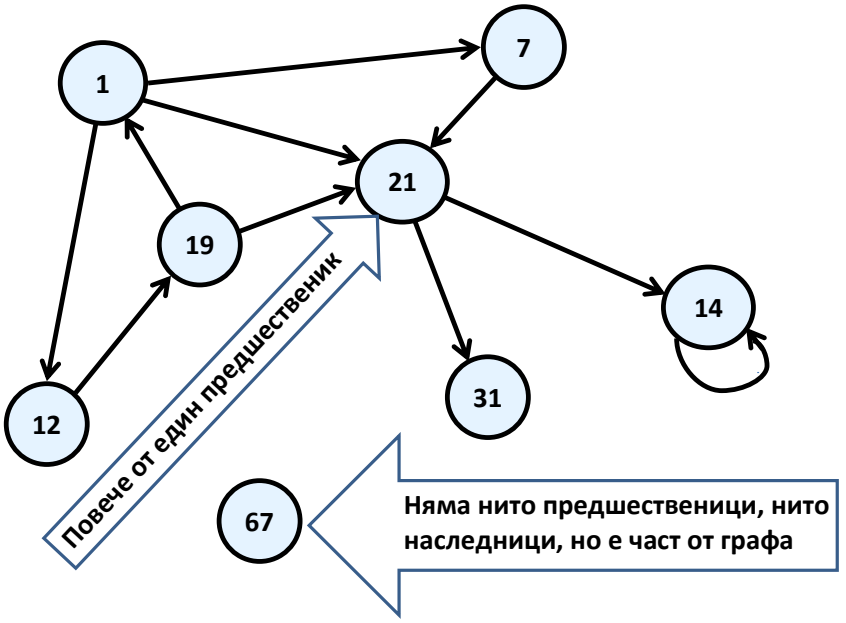
Графите се една изключително полезна и доста разпространена структура от данни. Използват се за описването на най-разнообразни взаимовръзки между обекти от практиката, свързани с почти всичко. Както ще видим по-късно, дървета са подмножество на графите, т.е. графите представляват една обобщена структура, позволяваща моделирането на доста голяма съвкупност от реални ситуации.

Честата употреба на графите в практиката е довела до задълбочени изследвания в "теория на графите", в която са известни огромен брой задачи за графи и за повечето от тях има и добре известно решение.

Графи – основни понятия

В тази точка ще въведем някои от по-важните понятия и дефиниции. Част от тях са аналогични на тези, въведени при структурата от данни дърво, но двете структури, както ще видим, имат много сериозни различия, тъй като дървото е само един частен случай на граф.

Да разгледаме следният примерен граф, чийто тип по-късно ще наречем краен ориентиран. В него отново имаме номерация на върховете, която е абсолютно произволна и е добавена, за да може по-лесно да говорим за някой конкретен:



Кръгчетата на схемата, ще наричаме **върхове**, а стрелките, които ги свързват, ще наричаме **ориентирани ребра (дъги)**. Върхът, от който излиза стрелката ще наричаме **предшественик** на този, който стрелката сочи. Например "19" е предшественик на "1". "1" от своя страна се явява **наследник** на "19". За разлика от структурата дърво, сега всеки един връх може да има повече от един предшественик. Например "21" има трима - "19", "1" и "7". Ако два върха са свързани с ребро, то казваме, че тези два върха са **инцидентни** с това ребро.

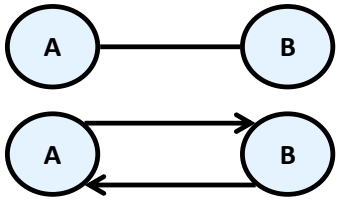
Следва дефиниция за **краен ориентиран граф (finite directed graph)**:

Краен ориентиран граф се нарича наредената двойката двойка **(V, E)**, където **V** е крайно множество от върхове, а **E** е крайно множество от ориентирани ребра. Всяко ребро ***e*** принадлежащо на **E** представлява наредена двойка от върхове ***u*** и ***v*** т.е. ***e*=(*u*, *v*)**, които еднозначно го определят.

За по-доброто разбиране на тази дефиниция силно препоръчваме на читателя да си мисли за върховете например като за градове, а ориентираните ребра като еднопосочни пътища. Така, ако единият връх е София, а другият е Велико Търново то еднопосочният път (дъгата) ще се нарича София-Велико Търново. Всъщност това е един от класическите примери за приложение на графите – в задачи свързани с пътища.

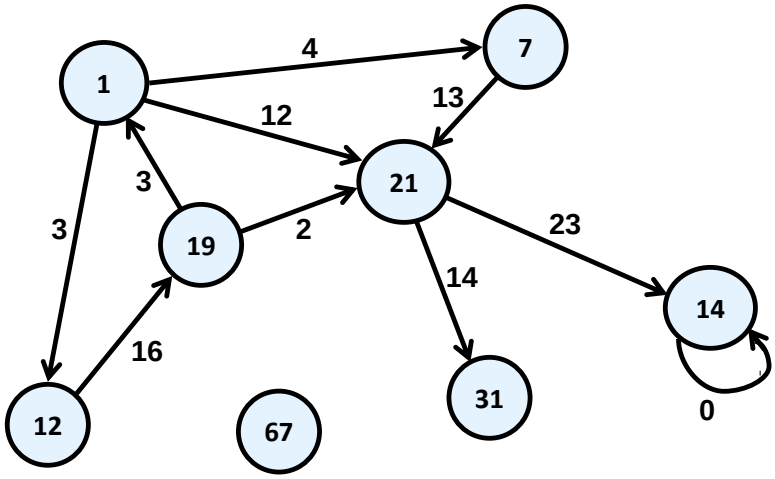
Ако вместо стрелки върховете са свързани с отсечки (както при структурата дърво), то тогава отсечките ще наричаме **неориентирани ребра**, а графът – **неориентиран**. На практика можем да си представяме, че едно неориентирано ребро от връх A до връх B представлява двупосочно ребро

еквивалентно на две противоположни ориентирани ребра между същите два върха:



Два върха свързани с ребро, ще наричаме **съседни**.

За ребрата може се зададе функция, която на всяко едно ребро съпоставя реално число. Тези така получени реални числа ще наричаме **тегла**. Като примери за тегла можем да дадем дължината на директните връзки между два съседни града, пропускателната способност на една тръба и др. Граф, който има тегла по ребрата се нарича **претеглен (weighted)**. Ето как се изобразява претеглен граф:



Път в граф ще наричаме последователност от върхове v_1, v_2, … , v_n, такава, че съществува ребро от v_i до v_{i+1} за всяко i от 1 до n-1. В нашия граф път е например последователността "1", "12", "19", "21". "7", "21" и "1" обаче не е път, тъй като не съществува ребро започващо от "21" и завършващо в "1".

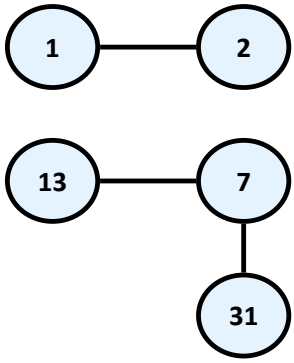
Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път "1", "12", "19", "21" е три.

Цена на път в претеглен граф, ще наричаме сумата от теглата на ребрата участващи в пътя. В реалния живот пътят от София до Варна например е равен на дължината на пътя от София до Велико Търново плюс дължината на пътя от Велико Търново до Варна. В нашия пример дължината на пътя "1", "12", "19" и "21" е равна на 3 + 16 + 2 = 21.

Цикъл е път, в който началният и крайният връх на пътя съвпадат. Пример за цикъл е "1", "12" и "19". "1", "7" и "21" обаче не е цикъл.

Примка ще наричаме ребро, което започва от и свършва в един и същ връх. В нашия пример върха "14" има примка.

Свързан неориентиран граф наричаме неориентиран граф, в който съществува път от всеки един връх до всеки друг. Например следният граф не е свързан, защото не съществува път от "1" до "7".



И така, вече имаме достатъчно познания, за да дефинираме понятието дърво по още един начин – като специален вид граф:

Дърво – неориентиран свързан граф без цикли.

Като леко упражнение оставяме на читателя да покаже защо двете дефиниции за дърво са еквивалентни.

Графи – видове представяния

Съществуват много различни начини за представяне на граф в програмирането. Различните представяния имат различни свойства и кое точно трябва да бъде избрано, зависи от конкретния алгоритъм, който искаме да приложим. С други думи казано – представяме графа си така, че операциите, които алгоритъмът ни най-често извършва върху него, да бъдат максимално бързи. Без да изпадаме в големи детайли ще изложим някои от най-често срещаните представяния на графи.

- **Списък на ребрата** – представя се, чрез списък от наредени двойки (v_i, v_j), където съществува ребро от $\mathbf{v_i}$ до $\mathbf{v_j}$. Ако графът е претеглен, то вместо наредена двойка имаме наредена тройка, като третият й елемент показва какво е теглото на даденото ребро.
- **Списък на наследниците** – в това представяне за всеки връх **v** се пази списък с върховете, към които сочат ребрата започващи от **v**. Тук отново, ако графът е претеглен, към всеки елемент от списъка с наследниците се добавя допълнително поле, показващо цената на реброто до него.

- **Матрица на съседство** – графът се представя като квадратна матрица **g[N][N]**, в която, ако съществува ребро от $\mathbf{v_i}$ до $\mathbf{v_j}$, то на позиция **g[i][j]** в матрицата е записано 1. Ако такова ребро не съществува, то в полето **g[i][j]** е записано 0. Ако графът е претеглен, в позиция **g[i][j]** се записва теглото на даденото ребро, а матрицата се нарича **матрица на теглата.** Ако между два върха в такава матрица не съществува път, то тогава се записва специална стойност, означаваща безкрайност.
- **Матрица на инцидентност между върхове и ребра –** в този случай отново се използва матрица, само че с размери **g[M][N]**, където M е броят на върховете, а N е броят на ребрата. Всеки стълб представя едно ребро, а всеки ред един връх. Тогава в стълба съответстващ на реброто ($\mathbf{v_i}$, $\mathbf{v_j}$) само и единствено на позиция **i** и на позиция **j** ще бъдат записани 1, а на останалите позиции в този стълб ще е записана 0. Ако реброто е примка т.е. е ($\mathbf{v_i}$, $\mathbf{v_i}$), то на позиция **i** записваме 2. Ако графът, който искаме да представим е ориентиран и искаме да представим ребро от $\mathbf{v_i}$ до $\mathbf{v_j}$, то на позиция **i** пишем 1, а на позиция **j** пишем -1.

Графи – основни операции

Основните операции в граф са:

- Създаване на граф
- Добавяне / премахване на връх / ребро
- Проверка дали даден връх / ребро съществува
- Намиране на наследниците на даден връх

Ще предложим примерна реализация на представяне на граф с матрица на съседство и ще покажем как се извършват повечето операции. Този вид реализация е удобен, когато максималният брой на върховете е предварително известен и когато той не е много голям (за да се реализира представянето на граф с N върха е необходима памет от порядъка на N^2 заради квадратната матрица). Поради това, няма да реализираме методи за добавяне / премахване на нов връх.

```
import java.util.LinkedList;
import java.util.List;

/**
 * Represents a directed unweighted graph structure.
 * @author Vesko Kolev
 */
public class Graph {
   // Contains the vertices of the graph
   private int vertices[][];
```

```
/**
 * Constructs the graph.
 * @param vertices - the vertices of the graph.
 */
public Graph(int[][] vertices) {
   this.vertices = vertices;
}

/**
 * Adds new edge from i to j.
 * @param i - the starting vertex.
 * @param j - the ending vertex.
 */
public void addEdge(int i, int j) {
   vertices[i][j] = 1;
}

/**
 * Removes the edge from i to j if such exists.
 * @param i - the starting vertex.
 * @param j - the ending vertex.
 */
public void removeEdge(int i, int j) {
   vertices[i][j] = 0;
}

/**
 * Checks whether there is an edge between vertex i and j.
 * @param i - the starting vertex.
 * @param j - the ending vertex.
 * @return true if there is an edge between
 * vertex i and vertex j.
 */
public boolean hasEdge(int i, int j) {
   return vertices[i][j] == 1;
}

/**
 * Returns the successors of a given vertex.
 * @param i - the vertex.
 * @return list with all successors of the given vertex.
 */
public List<Integer> getSuccessors(int i) {
   List<Integer> successors = new LinkedList<Integer>();

   for (int j = 0; j < vertices[i].length; i++) {
      if (vertices[i][j] == 1) {
```

```
            successors.add(j);
          }
        }

        return successors;
    }
}
```

Основни приложения и задачи за графи

Графите се използват за моделиране на много ситуации от реалността, а задачите върху графи моделират множество реални проблеми, които често се налага да бъдат решавани. Ще дадем само няколко примера:

- Карта на град може да се моделира с ориентиран претеглен граф. На всяка улица се съпоставя ребро с дължина съответстваща на дължината на улицата и посока – посоката на движение. Ако улицата е двупосочна може да й се съпоставят две ребра за двете посоки на движение. На всяко кръстовище се съпоставя връх. При такъв модел са естествени задачи като търсене на най-кратък път между две кръстовища, проверка дали има път между две кръстовища, проверка за цикъл (дали можем да се завъртим и да се върнем на изходна позиция), търсене на път с минимален брой завои и т.н.
- Компютърна мрежа може да се моделира с неориентиран граф, чиито върхове съответстват на компютрите в мрежата, а ребрата съответстват на комуникационните канали между компютрите. На ребрата могат да се съпоставят различни числа, примерно капацитет на канала или скорост на обмена и др. Типични задачи при такива модели на компютърна мрежа са проверка за свързаност между два компютъра, проверка за двусвързаност между две точки (съществуване на двойно-подсигурен канал, който остава при отказ на който и да е компютър) и др. В частност Интернет може да се моделира като граф, в който се решават задачи за маршрутизация на пакети, които се моделират като задачи за графи.
- Речната система в даден регион може да се моделира с насочен претеглен граф, в който всяка река се състои от едно или няколко ребра, а всеки връх съответства на място, където две или повече реки се вливат една в друга. По ребрата могат да се съпоставят стойности, свързани с количеството вода, което преминава по тях. Естествени при този модел са задачи като изчисление на обемите вода, преминаващи през всеки връх и предвиждане на евентуални наводнения при увеличаване на количествата.

Виждате, че графите могат да имат многобройни приложения. За тях има изписани стотици книги и научни трудове. Съществуват десетки класически задачи за графи, за които има известни решения или е известно, че нямат

ефективно решение. Ние няма да се спираме на тях. Надяваме се чрез краткото представяне да събудим интересът ви към темата и да ви подтикнем да отделите достатъчно внимание на задачите за графи от упражненията.

Упражнения

1. Да се напише програма, която намира броя на срещанията на дадено число в дадено дърво от числа.
2. Да се напише програма, която извежда корените на онези поддървета на дадено дърво, които имат точно **k** на брой върха, където **k** е дадено естествено число.
3. Да се напише програма, която намира броя на листата и броя на вътрешните върхове на дадено дърво.
4. Напишете програма, която по дадено двоично дърво от числа намира сумата на върховете от всяко едно ниво на дървото.
5. Да се напише програма, която намира и отпечатва всички върхове на двоично дърво, които имат за наследници само листа.
6. Да се напише програма, която проверява дали дадено двоично дърво е идеално балансирано.
7. Нека е даден граф **G(V, E)** и два негови върха **x** и **y**. Напишете програма, която намира най-краткия път между два върха по брой на върховете.
8. Нека е даден граф **G(V, E)**. Напишете програма, която проверява дали графът е цикличен.
9. Нека е даден граф **G(V, E)**. Напишете програма, която намира всички компоненти на свързаност на графа, т.е. намира всички негови максимални свързани подграфи. Максимален свързан подграф на **G** е свързан граф такъв, че няма друг подграф на **G**, който да е свързан и да го съдържа.
10. Нека е даден претеглен ориентиран граф **G(V, E)**, в който теглата по ребрата са неотрицателни числа. Напишете програма, която по зададен връх **x** от графа намира минималните пътища от него до всички останали.
11. Имаме N задачи, които трябва да бъдат изпълнени последователно. Даден е списък с двойки задачи, за които втората зависи от резултата от първата и трябва да бъде изпълнена след нея. Напишете програма, която подрежда задачите по такъв начин, че всяка задача да се изпълни след всички задачи, от които зависи. Ако не съществува такава наредба, да се отпечата подходящо съобщение.
12. Ойлеров цикъл в граф се нарича цикъл, който започва от даден връх, минава точно по веднъж през всички негови ребра и се връща в

началния връх. При това обхождане всеки връх може да бъде посетен многократно. Напишете програма, която по даден граф намира в него Ойлеров цикъл или установява, че такъв няма.

13. Хамилтонов цикъл в граф се нарича цикъл, съдържащ всеки връх в графа точно по веднъж. Да се напише програма, която при даден претеглен ориентиран граф **G(V, E)**, намира Хамилтонов цикъл с минимална дължина, ако такъв съществува.

Решения и упътвания

1. Обходете рекурсивно дървото в дълбочина и пребройте срещанията на даденото число.
2. Обходете рекурсивно дървото в дълбочина и проверете за всеки връх даденото условие.
3. Можете да решите задачата с рекурсивно обхождане на дървото в дълбочина.
4. Използвайте обхождане в дълбочина или в ширина и при преминаване от един връх в друг запазвайте в него на кое ниво се намира. Знаейки нивата на върховете търсената сума лесно се изчислява.
5. Можете да решите задачата с рекурсивно обхождане на дървото в дълбочина и проверка на даденото условие.
6. Чрез рекурсивно спускане в дълбочина за всеки връх на дървото изчислете дълбочините на лявото и дясното му поддърво. След това проверете непосредствено дали е изпълнено условието от дефиницията за идеално балансирано дърво.
7. Използвайте като основа алгоритъма за обхождане в ширина. Слагайте в опашката заедно с даден връх и неговия предшественик. Това ще ви помогне накрая да възстановите пътя между върховете (в обратен ред).
8. Използвайте обхождане в дълбочина или в ширина. Отбелязвайте за всеки връх дали вече е бил посетен. Ако в даден момент достигнете до връх, който е бил посетен по-рано, значи сте намерили цикъл.

   Помислете как можете да намерите и отпечатате самия цикъл. Ето една възможна идея: при обхождане в дълбочина за всеки връх пазите предшественика му. Ако в даден момент стигнете до връх, който вече е бил посетен, вие би трябвало да имате запазен за него някакъв път до началния връх. Текущият път в стека на рекурсията също е път до въпросния връх. Така в даден момент имаме два различни пътя от един връх до началния връх. От двата пътя лесно можете да намерите цикъл.
9. Използвайте като основа алгоритъма за обхождане в ширина или в дълбочина.
10. Използвайте алгоритъма на Dijkstra (намерете го в Интернет).

11. Търсената наредба се нарича "топологично сортиране на ориентиран граф". Може да се реализира по два начина:

    За всяка задача **t** пазим от колко на брой други задачи **P(t)** зависи. Намираме задача t_0, която не зависи от никоя друга ($P(t_0)=0$) и я изпълняваме. Намаляваме P(t) за всяка задача t, която зависи от t_0. Отново търсим задача, която не зависи от никоя друга и я изпълняваме. Повтаряме докато задачите свършат или до момент, в който няма нито една задача t_k с $P(t_k)=0$.

    Можем да решим задачата чрез обхождане в дълбочина на графа и печатане на всеки връх при напускането му. Това означава, че в момента на отпечатването на дадена задача всички задачи, които зависят от нея са били вече отпечатани.

12. За да съществува Ойлеров цикъл в даден граф, трябва графът да е свързан и степента на всеки негов връх да е четно число. Чрез поредица впускания в дълбочина можете да намирате цикли в графа и да премахвате ребрата, които участват в тях. Накрая като съедините циклите един с друг ще получите Ойлеров цикъл.

13. Ако напишете вярно решение на задачата, проверете дали работи за граф с 200 върха. Не се опитвайте да решите задачата, така че да работи бързо за голям брой върхове. Ако някой успее да я реши, ще остане трайно в историята!

Глава 18. Речници, хеш-таблици и множества

Автор

Владимир Цанев

В тази тема...

В настоящата тема ще разгледаме някои по-сложни структури от данни като речници и множества, и техните реализации с хеш-таблици и балансирани дървета. Ще обясним в детайли какво представляват хеширането и хеш-таблиците и защо са толкова важни в програмирането. Ще дискутираме понятието "колизия" и как се получават колизиите при реализация на хеш-таблици и ще предложим различни подходи за разрешаването им. Ще разгледаме абстрактната структура данни "множество" и ще обясним как може да се реализира чрез речник и чрез балансирано дърво. Ще дадем примери, които илюстрират приложението на описаните структури от данни в практиката.

Структура от данни "речник"

В предните няколко теми се запознахме с някои класически и много важни структури от данни – масиви, списъци и дървета. В тази секция ще се запознаем с така наречените "**речници**" (**dictionaries**), които са изключително полезни и широко използвани в програмирането.

Речниците са известни още като **асоциативни масиви** или **карти (maps)**. В тази тема ще използваме терминът "речник". Тези различни имена подчертават една и съща характеристика на тази структура от данни, а именно, че в тях всеки елемент представлява съответствие между ключ и стойност – наредена двойка. Аналогията идва от факта, че в един речник, например тълковния речник, за всяка дума (**ключ**) имаме обяснение (**стойност**). Подобни са тълкованията и на другите имена.

	**При речниците заедно с данните, които държим, пазим и ключ, по който ги намираме. Елементите на речниците са двойки (ключ, стойност), като ключът се използва при търсене.**

Структура от данни "речник" – пример

Ще илюстрираме какво точно представлява тази структура от данни с един конкретен пример от ежедневието.

Когато отидете на театър, опера или концерт често преди да влезете в залата или стадиона има гардероб, в който може да оставите дрехите си. Там давате дрехата си на служителката от гардероба, тя я оставя на определено място и ви дава номерче. След като свърши представлението, на излизане давате вашето номерче, и чрез него служителката намира точно вашата дреха и ви я връща.

Чрез този пример виждаме, че идеята да разполагаме с ключ (номерче, което ви дава служителката) за данните (вашата дреха) и да ги достъпваме чрез него, не е толкова нереална. В действителност това е подход, който се среща на много места, както в програмирането така и в много сфери на реалния живот.

При структурата речник този ключ може да на е просто номерче, а всякакъв друг обект. В случая, когато имаме ключ (номер), можем да реализираме такава структура като обикновен масив. Тогава множеството от ключове е предварително ясно – числата от **`0`** до **`n`**, където **`n`** е размерът на масива. Целта на речниците е да ни освободи, до колкото е възможно, от ограниченията за множеството на ключовете.

При речниците обикновено множеството от ключове е произволно множество от стойности, примерно реални числа или символни низове. Единственото задължително изискване е да можем да различим един ключ от друг. След малко ще се спрем по-конкретно на някои допълнителни изисквания към ключовете, необходими за различните реализации.

Речниците съпоставят на даден ключ дадена стойност. На един ключ може да се съпостави точно една стойност. Съвкупността от всички двойки (ключ, стойност) съставя речника.

Абстрактна структура данни "речник" (асоциативен масив, карта)

В програмирането абстрактната структура данни "речник" представлява съвкупност от наредени двойки (ключ, стойност), заедно с дефинирани операции за достъп до стойностите по ключ. Алтернативно тази структура може да бъде наречена още "карта" (map) или "асоциативен масив" (associative array).

Задължителни операции, които тази структура дефинира, са следните:

- **`Object put(key, value)`** – добавя в речника зададената наредена двойка. Ако вече имаме двойка с такъв ключ стойността за него се заменя с новата, а старата стойност се връща като резултат.

- **`Object get(key)`** – връща стойността по даден ключ. Ако в речника няма двойка с такъв ключ, връща **`null`**.
- **`boolean remove(key)`** – премахва стойността за този ключ от речника. Освен това връща дали е премахнат елемент от речника.

Ето и някои операции, които различните реализации на речници често предлагат:

- **`boolean isEmpty()`** – връща **`true`**, ако нямаме данни в речника и **`false`**, ако той съдържа поне една двойка (ключ, стойност).
- **`boolean contains(key)`** – връща **`true`**, ако в речникът има двойка с дадения ключ.
- **`int size()`** – връща броя елементи в речника.
- Други операции – например извличане на всички ключове, стойности или наредени двойки, в друга структура (масив, списък, множество), която лесно може да бъде обходена чрез цикъл.

Интерфейсът Map<K, V>

В Java има дефиниран стандартен интерфейс **`Map<K, V>`**, който дефинира всички основни операции, които речниците трябва да реализират. Този интерфейс съответства на абстрактната структура от данни "речник" и дефинира операциите, изброени по-горе, но без да предоставя конкретна реализация за всяка от тях.

В Java интерфейсите представляват спецификации за методите на даден клас. Те дефинират празни методи, които след това могат да бъдат имплементирани от конкретен клас, който обявява, че поддържа дадения интерфейс. Как работят интерфейсите и наследяването ще разгледаме подробно в главата "Принципи на обектно-ориентираното програмиране". За момента е достатъчно да знаете, че интерфейсите задават какви методи трябва да има в даден клас.

В настоящата тема ще разгледаме двата най-разпространени начина за реализация на речници – балансирано дърво и хеш-таблица. Изключително важно е да знаете, по-какво се отличават те един от друг и какви са основните принципи, свързани с тях. В противен случай рискувате да ги използвате неправилно и неефективно.

В Java има две важни имплементации на интерфейса **`Map`**: **`TreeMap`** и **`HashMap`**. **`TreeMap`** представлява имплементация с балансирано (червено-черно) дърво, а **`HashMap`** – имплементация с хеш-таблица.

Освен `HashMap` и `TreeMap` в Java има още имплементации на интерфейса `Map`, които обаче не трябва да се ползват освен, ако не ги познавате добре. Такива са например класовете `Hashtable`, `ConcurrentHashMap` и много други. Правилото е, че

	когато не знаете каква имплементация да ползвате винаги ползвайте `HashMap` или `TreeMap`.

От тази и следващата тема ще разберете в кои случаи да ползвате **`TreeMap<K, V>`** и в кои **`HashMap<K, V>`**.

Реализация на речник с червено-черно дърво

Тъй като имплементацията на речник чрез балансирано дърво е изключително сложна, няма да я разглеждаме във вид на сорс код. Вместо това ще разгледаме класа **`TreeMap<K, V>`**, който идва наготово заедно със стандартните библиотеки на Java.

Както беше обяснено вече в предната глава, червено-черното дърво е подредено двоично балансирано дърво за претърсване. Ето защо едно от важните изисквания, които са наложени върху множеството от ключове при използването на **`TreeMap<K, V>`**, е те **да имат наредба**. Това означава, че ако имаме два ключа, то или единият е по-голям от другия, или те са равни.

Използването на двоично дърво ни носи едно силно предимство: ключовете в речника се пазят сортирани. Благодарение на това свойство, ако данните ни трябват подредени по ключ, няма нужда да ги сортираме допълнително. Всъщност това свойство е единственото предимство на тази реализация пред реализацията с хеш-таблица. Пазенето на ключовете сортирани идва със своята цена. Работата с балансирани дървета е малко по-бавна от работата с хеш-таблици. По тази причина, ако няма специални изисквания за наредба на ключовете, за предпочитане е да се използва **`HashMap<K, V>`**.

	**Използвайте реализация на речник чрез балансирано дърво само когато се нуждаете от свойството наредените двойки винаги да са сортирани по ключ.**

Класът TreeMap<K, V>

Класът **`TreeMap<K, V>`** представлява имплементация на речник чрез червено-черно дърво. Този клас имплементира всички стандартни операции, типични за абстрактната структура данни речник и дефинирани в интерфейса **`Map<K, V>`**. В допълнение към тях **`TreeMap<K, V>`** дефинира още операции, свързани с наредбата на елементите:

- извличане на най-малък и най-голям елемент – **`firstEntry()`**, **`lastEntry()`**, **`firstKey()`**, **`lastKey()`**;
- извличане на всички елементи, по-малки или по-големи от даден ключ – **`headMap(key)`**, **`tailMap(key)`**;
- извличане на всички елементи в даден диапазон (примерно със стойност на ключа между 100 и 200) – **`subMap(startKey, endKey)`**.

Тези операции може да са много полезни при решавани на задачи, свързани с бързо извличане на подмножества на дадено множество.

Използване на класа TreeMap – пример

Сега ще решим един практически проблем, където използването на класа **`TreeMap`** е уместно. Нека имаме някакъв текст. Нашата задача ще бъде да намерим всички различни думи в текста, както и колко пъти се срещат всяка от тях в текста. Като допълнително условие искаме да изведем намерените думи по азбучен ред.

При тази задача използването на речник е особено подходящо. За ключове трябва да изберем думите от текста, а стойността записана в речника за всеки ключ ще бъде броят срещания на съответната дума.

Алгоритъмът за броене на думите се състои в следното: четем текста дума по дума и за всяка дума проверяваме дали вече присъства в речника. Ако отговорът е не, добавяме нов елемент в речника с ключ думата и стойност 1 (едно срещане). Ако отговорът е да, презаписваме стойността за текущата дума със старата ѝ стойност + 1 (увеличаваме с единица броя срещания на думата).

Използването на реализация на речник чрез балансирано дърво ни дава свойството, че когато обхождаме елементите му те ще бъдат сортирани по ключа. По този начин реализираме допълнително наложеното условие думите да са сортирани по азбучен ред. Следва реализация на описания алгоритъм:

TreeMapExample.java

```
import java.util.Map;
import java.util.TreeMap;
import java.util.Scanner;

/**
 * This class demonstrates using of {@link TreeMap} class.
 * @author Vladimir Tsanev
 */
public class TreeMapDemo {
   private static final String TEXT = "Test text words Count " +
         "words count teSt";
   public static void main(String[] args) {
      Map<String, Integer> wordsCounts = createWordsCounts(TEXT);
      printWordsCount(wordsCounts);
   }

   private static Map<String, Integer> createWordsCounts(
         String text) {
      Scanner textScanner = new Scanner(text);
```

```
      Map<String, Integer> words = new TreeMap<String, Integer>();
      while (textScanner.hasNext()) {
         String word = textScanner.next();
         Integer count = words.get(word);
         if (count == null) {
            count = 0;
         }
         words.put(word, count + 1);
      }
      return words;
   }

   private static void printWordsCount(
         Map<String, Integer> wordsCounts) {
      for (Map.Entry<String, Integer> wordEntry
            : wordsCounts.entrySet()) {
         System.out.printf(
            "word '%s' is seen %d times in the text%n",
            wordEntry.getKey(), wordEntry.getValue());
      }
   }
}
```

Изходът от примерната програма е следният:

```
word 'Count' is seen 1 times in the text
word 'Test' is seen 1 times in the text
word 'count' is seen 1 times in the text
word 'teSt' is seen 1 times in the text
word 'text' is seen 1 times in the text
word 'words' is seen 2 times in the text
```

В този пример за пръв път демонстрираме обхождане на всички елементи на речник – методът **printWordsCount(SortedMap<String, Integer>)**. За целта използваме подобрената версия на конструкцията за цикъл **for** (enhanced **for** loop). Начинаещите програмисти често срещат проблем при обхождане на речници, тъй като за разлика от списъците и масивите елементите на тази структура от данни са наредени двойки (ключ и стойност), а не просто единични обекти.

В примера използваме метода **entrySet()**, който ни връща множество с обекти, имплементиращи интерфейса **Map.Entry**. Този интерфейс декларира методите **getKey()** и **getValue()**, които ни дават достъп съответно до ключа и до стойността. Важно е да се разбере ясно, че самият речник не може да бъде обходен, но можем да обходим множество от неговите наредени двойки. Това разбира се не ни ограничава по никакъв начин.

Интерфейсът Comparable<K>

При използване на **TreeMap<K, V>** има задължително изискване ключовете да са от тип, чиито стойности могат да се сравняват по големина. В нашия пример ползваме за ключ обекти от тип **String**.

Класът **String** имплементира интерфейса **Comparable**, като сравнението е стандартно. Какво означава това? Тъй като низовете в Java са case sensitive (т.е. има разлика между главна и малка буква), то думи като "Count" и "count" се смятат за различни, а думите, които започват с малка буква, са след тези с голяма. Понякога това може да е неудобство, но то е следствие от естествената наредба на низовете дефинирана в класа **String**. Тази дефиниция идва от имплементацията на метода **compareTo(String)**, чрез който класът **String** имплементира интерфейса **Comparable**.

Интерфейсът Comparator<K>

Какво можем да направим, когато естествената наредба не ни удовлетворява? Например, ако искаме при сравнението на думите да не се прави разлика между малки и главни букви.

Един вариант е след като прочетем дадена дума да я преобразуваме към малки или главни букви. Този подход ще работи за символни низове, но понякога ситуацията е по-сложна. Затова сега ще покажем друго решение, което работи за всеки произволен клас, който няма естествена наредба (не имплементира **Comparable**) или има естествена наредба, но ние искаме да я променим.

За сравнение на обекти по изрично дефинирана наредба в Java е има един интерфейс **Comparator<E>**. Той дефинира функция за сравнение **compare(E o1, E o2)**, която задава алтернативна на естествената наредба. Нека разгледаме в детайли този интерфейс. За интерфейсите ще ви разкажем подробно в главата "Принципи на ООП". За момента приемете, че те представляват дефиниции на един или няколко метода, които могат да бъдат имплементирани от даден клас.

Когато създаваме обект от класа **TreeMap<K, V>** можем да подадем на конструктора му референция към **Comparator<K>** и той да използва него при сравнение на ключовете (които са елементи от тип **K**).

Ето една реализация чрез анонимен клас на интерфейса **Comparator<K>**, която решава проблема с главните и малките букви:

```
Comparator<String> caseInsensitiveComparator =
   new Comparator<String>(){
      @Override
      public int compare(String o1, String o2) {
         return o1.compareToIgnoreCase(o2);
      }
};
```

Нека използваме този **Comparator<E>** при създаването на речника:

```
Map<String, Integer> words =
   new TreeMap<String, Integer>(caseInsensitiveComparator);
```

След тази промяна резултатът от изпълнението на програмата ще бъде:

```
word 'Count' is seen 2 times in the text
word 'Test' is seen 2 times in the text
word 'text' is seen 1 times in the text
word 'words' is seen 2 times in the text
```

Виждаме, че за ключ остава вариантът на думата, който е срещнат за първи в текста. Това е така, тъй като при извикване на метода **put()** се подменя само стойността, но не и ключът.

Използвайки **Comparator<E>** ние на практика сменихме дефиницията за подредба на ключове в рамките на нашия речник. Ако за ключ използвахме клас, дефиниран от нас, примерно **Student**, който имплементира **Comparable<E>**, бихме могли да постигнем същия ефект чрез подмяна на реализацията на метода му **compareTo(Student)**. Има обаче едно изискване, което трябва винаги да се стремим да спазваме, когато имплементираме **Comparable<K>**. То гласи следното:

Винаги, когато два обекта са еднакви (`equals(Object)` връща `true`), `compareTo(E)` трябва да връща 0.

Удовлетворяването на това условие ще ни позволи да ползваме обектите от даден клас за ключове, както в реализация с балансирано дърво (**TreeMap**, конструиран без **Comparator**), така и в реализация с хеш-таблица (**HashMap**).

Хеш-таблици

Нека сега се запознаем със структурата от данни хеш-таблица, която реализира по един изключително ефективен начин абстрактната структура данни речник. Ще обясним в детайли как работят хеш-таблиците и защо са толкова ефективни.

Реализация на речник с хеш-таблица

Реализацията с хеш-таблица има важното предимство, че времето за достъп до стойност от речника, при правилно използване, не зависи от броя на елементите в него (поне на теория).

За сравнение да вземем списък с елементи, които са подредени в случаен ред. Искаме да проверим дали даден елемент се намира в него. В най-лошия случай, трябва да проверим всеки един елемент от него, за да дадем категоричен отговор на въпроса "съдържа ли списъкът елемента или не".

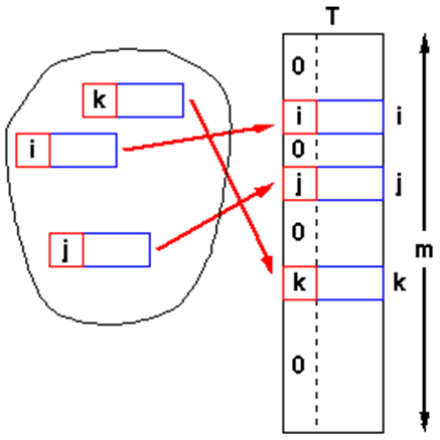
Очевидно е, че броят на тези сравнения зависи (линейно) от броя на елементите в списъка.

При хеш-таблиците, ако разполагаме с ключ, броят сравнения, които трябва да извършим, за да установим има ли стойност с такъв ключ, е константен и не зависи от броя на елементите в нея. Как точно се постига такава ефективност ще разгледаме в детайли по-долу.

Когато реализациите на някои структури от данни ни дават време за достъп до елементите й, независещ от броя на елементите в нея, се казва, че те притежават свойството **random access** (**свободен достъп**). Такова свойство обикновено се наблюдава при реализации на абстрактни структури от данни с хеш-таблици и масиви.

Какво е хеш-таблица?

Хеш-таблицата обикновено е реализирана с масив. Тя съдържа наредени двойки (ключ, стойност), които са разположени в масива на пръв поглед случайно и непоследователно. В позициите, в които нямаме наредена двойка, имаме празен елемент (`null`):

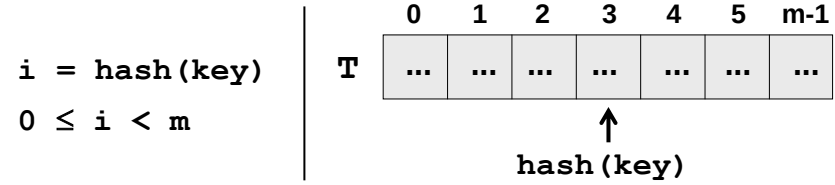


Размерът на таблицата (масива), наричаме **капацитет (capacity)** на хеш-таблицата. **Степен на запълненост** наричаме реално число между 0 и 1, което съответства на отношението между броя на запълнените елементи и текущия капацитет. На фигурата имаме хеш-таблица с 3 елемента и капацитет `m`. Степента на запълване на хеш-таблицата е `3/m`.

Добавянето и търсенето на елементи става, като върху ключа се приложи някаква функция `hash(key)`, която връща число, наречено **хеш-код**. Като вземем остатъка при деление на този хеш-код с капацитета `m` получаваме число между `0` и `m-1`:

```
index = hash(key) % m
```

На фигурата е показана хеш-таблица `T` с капацитет `m` и хеш-функция `hash(key)`:



Това число ни дава позицията, на която да търсим или добавяме наредената двойка. Ако хеш-функцията разпределя ключовете равномерно, в болшинството случаи на различен ключ ще съответства различна хеш-стойност и по този начин във всяка клетка от масива ще има най-много един ключ. В крайна сметка получаваме изключително бързо търсене и бързо добавяне. Разбира се, може да се случи различни ключове да имат един и същ хеш-код. Това е специален случай, който ще разгледаме след малко.

Използвайте реализация на речник чрез хеш-таблици, когато се нуждаете от максимално бързо намиране на стойностите по ключ.

Капацитетът на таблицата се увеличава, когато броят на наредените двойки в хеш-таблицата стане равен или по-голям от дадена константа, наречена **максимална степен на запълване (load factor)**. При разширяване на капацитета (най-често удвояване) всички елементи се преподреждат според своя хеш-код и стойността на новия капацитет. Степента на запълване след преподреждане значително намалява. Операцията е времеотнемаща, но се извършва достатъчно рядко, за да не влияе на цялостната производителност на операцията добавяне.

Класът HashMap<K, V>

Класът `HashMap<K, V>` е стандартна имплементация на речник с хеш-таблица в Java Collections Framework. Ще се спрем на основните операции, които той предоставя, както и на един конкретен пример, който илюстрира използването на класа и неговите методи.

Основни операции с класа HashMap<K, V>

Създаването на хеш-таблица става чрез извикването на някои от конструкторите на `HashMap<K, V>`. Чрез тях можем да зададем начални стойности за капацитет и максимална степен на запълване. Добре е, ако предварително знаем приблизителният брой на елементите, които ще бъдат добавени в нашата хеш-таблица, да го укажем още при създаването ѝ. Така ще избегнем излишното разширяване на таблицата и ще постигнем по-добра ефективност. По подразбиране стойността на началния капацитет 16, а на максималната степен на запълване е 0.75.

Да разгледаме какво прави всеки един от методите реализирани в класа **HashMap<K, V>**:

- **V put(K, V)** добавя нова стойност за даден ключ или презаписва вече съществуващата за този ключ. В резултат се връща старата стойност за посочения ключ или **null**, ако няма стара стойност. Операцията работи изключително бързо.
- **void putAll(Map<K, V>)** добавя всички наредени двойки от друг речник в текущия. Извикването на този метод е еквивалентно на извикването на **put(K, V)** за всеки един елемент на речника, който е подаден като параметър.
- **V get(Object)** връща стойността за дадения ключ или **null**, ако няма елемент с такъв ключ. Операцията работи изключително бързо.
- **V remove(K)** изтрива от речника елемента с този ключ. Операцията работи изключително бързо.
- **void clear()** премахва всички елементи от речника.
- **boolean containsKey(K)** проверява дали в речника присъства наредена двойка с посочения ключ. Операцията работи изключително бързо.
- **boolean containsValue(V)** проверява дали в речника присъстват една или повече наредени двойки с посочената стойност. Тази операция работи бавно, тъй като проверява всеки елемент на хеш-таблицата.
- **boolean isEmpty()** връща **true** ако в речника няма нито една наредена двойка и **false** в противен случай.
- **int size()** връща броя на наредените двойки в речника.
- **Set<Map.Entry<K, V> entriesSet()** връща множество от наредените двойки в речника. Така можем лесно да ги обходим в цикъл.
- **Set<K> keySet()** връща множество от всички ключове в речника.
- **Collection<V> values()** връща колекция (може да има повторения) от всички стойности в речника.

Студенти и оценки – пример

Сега ще илюстрираме как се ползват някои от описаните по-горе операции чрез един пример. Имаме студенти. Всеки от тях би могъл да има най-много една оценка. Искаме да съхраняваме оценките в някаква структура, в която можем бързо да търсим по име на студент.

За тази задача ще създадем хеш-таблица с начален капацитет 6. Тя ще има за ключове имената на студентите, а за стойности – някакви техни оценки. Добавяме 6 примерни студента, след което наблюдаваме какво се случва като отпечатваме на стандартния изход техните данни. Ето как изглежда кодът от този пример:

```
Map<String, Double> studentsNarks =
   new HashMap<String, Double>(6);
studentsNarks.put("Pesho", 3.00);
studentsNarks.put("Gosho", 4.50);
studentsNarks.put("Nakov", 5.50);
studentsNarks.put("Vesko", 3.50);
studentsNarks.put("Tsanev", 4.00);
studentsNarks.put("Nerdy", 6.00);

Double tsanevMark = studentsNarks.get("Tsanev");
System.out.printf("Tsanev's mark: %.2f %n", tsanevMark);

studentsNarks.remove("Tsanev");
System.out.println("Tsanev removed.");

System.out.printf("Is Tsanev in the hash table: %b %n",
   studentsNarks.containsKey("Tsanev"));

studentsNarks.put("Nerdy", 3.25);
System.out.println("Nerdy's mark changed.");

System.out.println("Students and marks:");

for (Map.Entry<String, Double> studentMark
      : studentsNarks.entrySet()) {
   System.out.printf("%s has %.2f%n",
      studentMark.getKey(), studentMark.getValue());
}
System.out.printf("There are %d students.%n",
   studentsNarks.size());
studentsNarks.clear();
System.out.println("Students hashmap cleared.");
System.out.printf("Is hash table empty: %b%n",
   studentsNarks.isEmpty());
```

Изходът от изпълнението на този код е следният:

```
Tsanev's mark: 4,00
Tsanev removed.
Is Tsanev in the hash table: false
Nerdy's mark changed.
Students and marks:
Nerdy has 3,25
Nakov has 5,50
Vesko has 3,50
Gosho has 4,50
Pesho has 3,00
```

```
There are 5 students.
Students hashmap cleared.
Is hash table empty: true
```

Виждаме, че редът, в който се отпечатват студентите е напълно случаен. Това е така, защото при хеш-таблиците (за разлика от балансираните дървета) елементите не се пазят сортирани. Дори ако текущият капацитет на таблицата се промени докато работим с нея, много е вероятно да се промени и редът, в който се пазят наредените двойки. На причината за това поведение обаче ще се спрем по-долу.

Важно е да се запомни, че при хеш-таблиците не можем да разчитаме на никаква наредба на елементите. Ако се нуждаем от такава, можем преди отпечатване да сортираме елементите. Друг вариант е да използваме **`TreeMap<K, V>`**.

Хеш-функции и хеширане

Сега ще се спрем по-детайлно на понятието, хеш-код, което употребихме малко по-рано. Хеш-кодът представлява числото, което ни връща т.нар. **хеш-функция**, приложена върху ключа. Това число трябва да е различно за всеки различен ключ или поне с голяма вероятност при различни ключове хеш-кодът трябва да е различен.

Хеш-функции

Съществува понятието **перфектна хеш-функция** (perfect hash function). Това означава, че ако имаме N ключа, тази функция на всеки ключ ще съпоставя различно цяло число в някакъв смислен интервал (например от 0 до N-1). Намирането на такава функция в общия случай е доста трудна, почти невъзможна задача. Такива функции си струва да се използват само при множества от ключове, които са с предварително известни елементи или ако множеството от ключове поне рядко се променя.

В практиката се използват други, не чак толкова "перфектни" хеш-функции. Сега ще разгледаме няколко примера за хеш-функции, които се използват директно в Java библиотеките.

Методът hashCode() в Java платформата

Всички Java класове имат метод **`hashCode()`**, който връща стойност от тип **`int`**. Този метод се наследява от класа **`Object`**, който стои в корена на йерархията на всички Java класове.

Имплементацията в класа **`Object`** на класа **`hashCode()`** е **`native`** метод (метод имплементиран на ниско ниво от доставчика на виртуалната машина), който обикновено връща число базирано на адреса на обекта в паметта, но това въобще не е задължително. Тъй като този метод е имплементиран от

създателя на виртуалната машина, не се знае каква точно ще е имплементацията. Връщаната стойност от този метод е непредсказуема и затова никога не трябва да разчитате на нея.

Друг пример за хеш-функция, която идва директно от Java, е тази, която се ползва от класовете, дефиниращи цели числа като, **Integer**, **Byte** и **Short**. Там за хеш-код се ползва стойността на самото число.

Да разгледаме един по-сложен пример за хеш-функция, който също идва от вградените в Java класове. Става въпрос за имплементацията на хеш-функция, която се ползва от класа **String**. Тя връща 0, ако низът е празен, а в противен случай хеш-кодът се изчислява по формулата:

$$hash(s) = s_0*31^{n-1} + s_1*31^{n-2} + \ldots + s_n$$

където s_i, е i-ият символ на низа, а **n** е неговата дължина.

На читателя оставяме да разгледа други имплементации на метода **hashCode()** в някои от най-често използваните класове като **Date**, **Long**, **Float** и **Double**.

Сега, нека се спрем на въпроса как да имплементираме сами този метод за нашите класове. Вече обяснихме, че оставянето на имплементацията, която идва наготово от **Object**, не е допустимо решение. Друга много проста имплементация е винаги да връщаме някаква фиксирана константа, примерно:

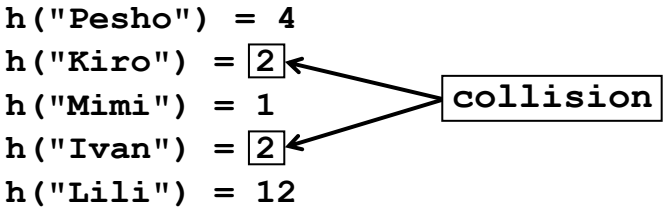
```
@Override
public int hashCode() {
   return 53;
}
```

Ако използваме хеш-таблица и ползваме за ключовете й обекти от клас, който има горната имплементация на **hashCode()**, ще получим много лоша производителност, защото всеки път, когато добавяме нов елемент в таблицата, ще трябва да го слагаме на едно и също място. Когато търсим, всеки път ще попадаме в една и съща клетка на таблицата.

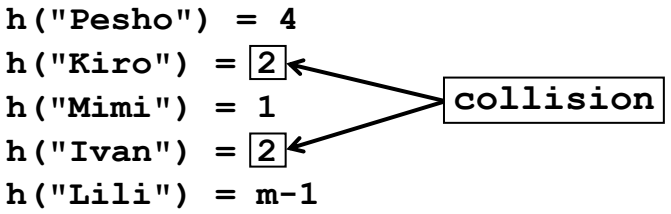
За да се избягва описаното неблагоприятно поведение, трябва хеш-функцията да разпределя ключовете максимално равномерно сред възможните стойности за хеш-код.

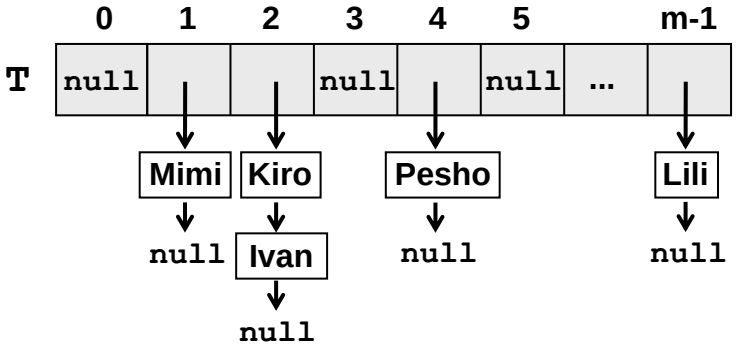
Колизии при хеш-функциите

Ситуация, при която два различни ключа връщат едно и също число за хеш-код наричаме **колизия**:



Как да решим проблема с колизиите ще разгледаме подробно в следващия параграф. Най-простото решение, обаче е очевидно: двойките, които имат ключове с еднакви хеш-кодове да нареждаме в списък:





Следователно при използване на константа за хеш-код, нашата хеш-таблица се изражда в линеен списък и употребата ѝ става неефективна.

Имплементиране на метода hashCode()

Ще дадем един стандартен алгоритъм, по който можем сами да имплементираме **hashCode()**, когато ни се наложи:

Първо трябва да определим полетата на класа, които участват по някакъв начин в имплементацията на **equals()** метода. Това е необходимо, тъй като винаги, когато **equals()** е **true** трябва резултатът от **hachCode()** да е един и същ. Така полетата, които не участват в пресмятането на **equals()**, не трябва да участват и в изчисляване на **hashCode()**.

След като сме определили полетата, които ще участват в изчислението на **hashCode()**, трябва по някакъв начин да получим за тях стойности от тип **int**. Ето една примерна схема:

- Ако полето е **boolean**, за **true** взимаме **1**, а за **false** взимаме **0**.
- Ако полето е от тип **int**, **byte**, **short**, **char** можем да го преобразуваме към **int**, чрез оператора за явно преобразуване (**int**). Ако е от тип **long**, го разделяме на 2 части по 32 бита и получаваме от него две **int** стойности.
- Ако полето е от тип **float** или **double**, можем да го превърнем в целочислен вид чрез методите **Float.floatToIntBits()** или **Double.doubleToLongBits()**. В случая с **double** резултатът третираме както **long** от горната точка.
- Ако полето не е от примитивен тип, просто извикваме метода **hashCode()** на този обект. Ако стойността на полето е **null**, връщаме **0**.
- Ако полето е масив или някаква колекция, извличаме хеш-кода за всеки елемент на тази колекция.

Накрая сумираме получените **int** стойности, като преди всяко събиране умножаваме временния резултат с някое просто число (например 31), като игнорираме евентуалните препълвания на типа **int**.

В крайна сметка получаваме хеш-код, който е добре разпределен в пространството от всички 32-битови стойности. Можем да очакваме, че при така изчислен хеш-код колизиите ще са рядкост, тъй като всяка промяна в някое от полетата, участващи в описаната схема за изчисление, води до съществена промяна в хеш-кода.

Имплементиране на hashCode() – пример

Да илюстрираме горният алгоритъм с един пример. Нека имаме клас, чиито обекти представляват точка в тримерното пространство. И нека точката вътрешно представяме чрез нейните координати по трите измерения x, y и z:

Point3D.java

```
/**
 * Class representing points in three dimensional space.
 * @author Vladimir Tsanev
 */
public class Point3D {
   private double x;
   private double y;
   private double z;

   /**
    * Construct new {@link Point3D} instance by specified
    * Cartesian coordinates of the point.
```

```
    * @param x - x coordinate  of the point
    * @param y - y coordinate  of the point
    * @param z - z coordinate  of the point
    */
   public Point3D(double x, double y, double z) {
      super();
      this.x = x;
      this.y = y;
      this.z = z;
   }
}
```

Можем лесно да реализираме **hashCode()** по описания по-горе алгоритъм.

Автоматично генериране на hashCode() в Eclipse

Eclipse, както и повечето модерни среди за разработка, могат автоматично да генерират кода за методите **equals()** и **hashCode()**. При Eclipse имплементацията на **hashCode()** ще бъде по алгоритъм, сходен на описания по-горе. Можете да генерирате автоматично методите **equals()** и **hashCode()** за даден клас по следния начин: от менюто **Source** избирате **Generate hasCode and equals()...** След това избирате полетата, които искате да участват в изчисленията за двата метода и натискате бутона [OK]. За нашия клас **Point3D** генерираният код е следният:

```
@Override
public int hashCode() {
   final int prime = 31;
   int result = 1;
   long temp;
   temp = Double.doubleToLongBits(x);
   result = prime * result + (int) (temp ^ (temp >>> 32));
   temp = Double.doubleToLongBits(y);
   result = prime * result + (int) (temp ^ (temp >>> 32));
   temp = Double.doubleToLongBits(z);
   result = prime * result + (int) (temp ^ (temp >>> 32));
   return result;
}

@Override
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
```

```
    Point3D other = (Point3D) obj;
    if (Double.doubleToLongBits(x)
          != Double.doubleToLongBits(other.x))
       return false;
    if (Double.doubleToLongBits(y)
          != Double.doubleToLongBits(other.y))
       return false;
    if (Double.doubleToLongBits(z)
          != Double.doubleToLongBits(other.z))
       return false;
    return true;
}
```

Тази имплементация е несравнимо по-добра, от това да не правим нищо или да връщаме константа. Въпреки това колизиите и при нея се срещат, но доста по-рядко.

Решаване на проблема с колизиите

На практика колизиите могат да се избегнат в изключително редки и специфични ситуации. За това е необходимо да живеем с идеята за тяхното присъствие в нашите хеш таблици и да се съобразяваме с тях. Нека разгледаме няколко стратегии за справяне с колизиите:

Нареждане в списък (chaining)

Най-разпространеният начин за решаване на проблема с колизиите е нареждането в списък (chaining). Той се състои в това двойките ключ и стойност, които имат еднакъв хеш-код за ключа да се нареждат в списък един след друг.

Реализация на речник чрез хеш-таблица и chaining

Нека си поставим за задача да реализираме структурата от данни речник чрез хеш-таблица с решаване на колизиите чрез нареждане в списък (cahining). Да видим как може да стане това. Първо ще дефинираме клас, който описва наредена двойка (**entry**). Той капсулира в себе си двойка ключ и стойност:

DictionaryEntry.java

```
/**
 * This class is used by Dictionary Abstract Data Type (ADT).
 * It encapsulates Key and Value objects.
 * @author Vladimir Tsanev
 * @param <K> - type of the keys.
 * @param <V> - type of the values.
 */
```

```
public class DictionaryEntry<K, V> {
   private K key;
   private V value;

   public DictionaryEntry(K key, V value) {
      this.key = key;
      this.value = value;
   }

   public K getKey() {
      return this.key;
   }

   public V getValue() {
      return this.value;
   }

   @Override
   public String toString() {
      return String.format("[%s, %s]", key, value);
   }
}
```

Този клас има конструктор, който приема ключ и стойност. Дефинирани са два метода за достъп съответно за ключа (**getKey()**) и стойността (**getValue()**). Ще отбележим, че нарочно нямаме публични методи, чрез които да променяме стойностите на ключа и стойността. Това прави този клас непроменяем (immutable). Това е добра идея, тъй като обектите, които ще се пазят вътрешно в реализациите на речника, ще бъдат същите като тези, които ще връщаме например при реализацията на метод за вземане на всички наредени двойки.

Предефинирали сме метода **toString()**, за да можем лесно да отпечатваме наредената двойка на стандартния изход или в текстов поток.

Следва примерен шаблонен интерфейс, който дефинира най-типичните операции за типа речник:

Dictionary.java

```
/**
 * Interface that defines basic methods needed
 * for a class which maps keys to values.
 * @param <K> - type of the keys
 * @param <V> - type of the values
 * @author Vladimir Tsanev
 */
public interface Dictionary<K, V>
```

```
    extends Iterable<DictionaryEntry<K, V>> {

  /**
  * Adds specified value by specified key to the dictionary.
   * If the key already exists its value is replaced with the
   * new value and the old value is returned.
   * @param key - key for the new value
   * @param value - value to be mapped with that key
   * @return the old value for the specified key or null if the
   *    key does not exists
   * @throws NullPointerException if specified key is null.
   */
  public V put(K key, V value);

  /**
   * Finds the value mapped by specified key.
   * @param key - key for which the value is needed.
   * @return value for the specified key if present,
   *         or null if there is no value with such key.
   */
  public V get(K key);

  /**
   * Removes a value mapped by specified key.
   * @param key - key for which the value will be removed
   * @return <code>true</code> if value for the specified
   *    key if present, or <code>false</code> if there is
   *    no value with such key in the dictionary.
   */
  public boolean remove(K key);

  /**
   * Checks if there are any elements in the dictionary.
   * @return <code>true</code> if there is more than
   *    one element in the dictionary, and
   *    <code>false</code> otherwise.
   */
  public boolean isEmpty();

  /**
   * Removes all elements from the dictionary.
   */
  public void clear();
}
```

В интерфейса по-горе, както и в предходния клас използваме шаблонни типове (generics), чрез които декларираме параметри за типа на ключовете (**K**) и типа стойностите (**V**). Това позволява нашият речник да бъде

използват с произволни типове за ключовете и за стойностите. Единственото изискване е ключовете да дефинират коректно методите **equals()** и **hashCode()**.

Нашият интерфейс **Dictionary<K, V>** прилича много на интерфейса **Map<K, V>**, но е по-прост от него и описва само най-важните операции върху типа данни "речник". Той наследява системния интерфейс **Iterable <DictionaryEntry<K, V>>**, за да позволи речникът да бъде обхождан във for цикъл.

Следва примерна имплементация на речник, в който проблемът с колизиите се решава чрез нареждане в списък (chaining):

HashDictionary.java

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

/**
 * Implementation of {@link Dictionary} interface
 * using hash table. Collisions are resolved by chaining.
 * @author Vladimir Tsanev
 * @param <K> - the type of the keys
 * @param <V> - the type of the values
 */
public class HashDictionary<K, V> implements Dictionary<K, V> {
   private static final int DEFAULT_CAPACITY = 2;
   private static final float DEFAULT_LOAD_FACTOR = 0.75f;
   private List<DictionaryEntry<K, V>>[] table;
   private float loadFactor;
   private int threshold;
   private int size;

   public HashDictionary() {
      this(DEFAULT_CAPACITY, DEFAULT_LOAD_FACTOR);
   }

   @SuppressWarnings("unchecked")
   private HashDictionary(int capacity, float loadFactor) {
      this.table = new List[capacity];
      this.loadFactor = loadFactor;
      this.threshold =
         (int) (this.table.length * this.loadFactor);
   }

   @Override
```

```
public void clear() {
   Arrays.fill(this.table, null);
   this.size = 0;
}

private List<DictionaryEntry<K, V>> findChain(
      K key, boolean createIfMissing) {
   int index = key.hashCode();
   index = index % this.table.length;
   if (table[index] == null && createIfMissing) {
      table[index] = new ArrayList<DictionaryEntry<K, V>>();
   }
   return table[index];
}

@Override
public V get(K key) {
   List<DictionaryEntry<K, V>> chain = findChain(key, false);
   if (chain != null) {
      for (DictionaryEntry<K, V> dictionaryEntry : chain) {
         if (dictionaryEntry.getKey().equals(key)) {
            return dictionaryEntry.getValue();
         }
      }
   }
   return null;
}

@Override
public boolean isEmpty() {
   return size == 0;
}

@Override
public V put(K key, V value) {
   List<DictionaryEntry<K, V>> chain = findChain(key, true);
   for (int i=0; i<chain.size(); i++) {
      DictionaryEntry<K, V> entry = chain.get(i);
      if (entry.getKey().equals(key)) {
         // Key found -> replace its value with the new value
         DictionaryEntry<K, V> newEntry =
            new DictionaryEntry<K, V>(key, value);
         chain.set(i, newEntry);
         return entry.getValue();
      }
   }
   chain.add(new DictionaryEntry<K, V>(key, value));
   if (size++ >= threshold) {
```

```
      expand();
   }
   return null;
}

/**
 * Expands the underling table
 */
@SuppressWarnings("unchecked")
private void expand() {
   int newCapacity = 2 * this.table.length;
   List<DictionaryEntry<K, V>>[] oldTable = this.table;
   this.table = new List[newCapacity];
   this.threshold = (int) (newCapacity * this.loadFactor);
   for (List<DictionaryEntry<K, V>> oldChain : oldTable) {
      if (oldChain != null) {
         for (DictionaryEntry<K, V> dictionaryEntry : oldChain){
            List<DictionaryEntry<K, V>> chain =
               findChain(dictionaryEntry.getKey(), true);
            chain.add(dictionaryEntry);
         }
      }
   }
}

@Override
public boolean remove(K key) {
   List<DictionaryEntry<K, V>> chain = findChain(key, false);
   if (chain != null) {
      for (int i=0; i<chain.size(); i++) {
         DictionaryEntry<K, V> entry = chain.get(i);
         if (entry.getKey().equals(key)) {
            // Key found -> remove it
            chain.remove(i);
            return true;
         }
      }
   }
   return false;
}

@Override
public Iterator<DictionaryEntry<K, V>> iterator() {
   List<DictionaryEntry<K, V>> entries =
      new ArrayList<DictionaryEntry<K, V>>(this.table.length);
   for (List<DictionaryEntry<K, V>> chain : this.table) {
      if (chain != null) {
         entries.addAll(chain);
```

```
            }
        }
        return entries.iterator();
    }
}
```

Ще обърнем внимание на по-важните моменти в този код. Нека започнем от конструктора. Единственият публичен конструктор е конструкторът по подразбиране. Той в себе си извиква друг конструктор като му подава някакви предварително зададени стойности за капацитет и степен на запълване. На читателя предоставяме да реализира валидация на тези параметри и да направи и този конструктор публичен, за да предостави повече гъвкавост на ползвателите на този клас.

Следващото нещо, на което ще обърнем внимание, е това как е реализирано нареждането в списък. При конструирането на хеш-таблицата в конструктора инициализираме масив от списъци, които ще съдържат нашите **DictionaryEntry** обекти. За вътрешно ползване сме реализирали един метод **findChain()**, който изчислява хеш-кода на ключа като вика метода **hashCode()** и след това разделя върнатата хеш-стойност на дължината на таблицата (капацитета). Така се получава индексът на текущия ключ в масива, съхраняващ елементите на хеш-таблицата. Списъкът с всички елементи, имащи съответния хеш-код се намира в масива на изчисления индекс. Ако списъкът е празен, той има стойност **null**. В противен случай в съответната позиция има списък от елементи за съответния ключ.

На метода **findChain()** се подава специален параметър, който указва дали да създава празен списък, ако за подадения ключ все още няма списък с елементи. Това предоставя удобство на методите за добавяне на елементи и за преоразмеряване на хеш-таблицата.

Другото нещо, на което ще обърнем внимание, е методът **expand()**, който разширява текущата таблица, когато се достигне максималното допустимо запълване. За целта създаваме нова таблица (масив), двойно по-голяма от старта. Изчисляваме новото максимално допустимо запълване, това е полето **threshold**. Следва най-важната част. Разширили сме таблицата и по този начин сме сменили стойността на **this.table.length**. Ако потърсим някой елемент, който вече сме добавили, методът **findChain(K key)**, изобщо няма да върне правилната верига, в която да го търсим. Затова се налага всички елементи от старата таблица да се прехвърлят, като не просто се копират веригите, а се добавят наново обектите от клас **DictionaryEntry** в новосъздадени вериги.

За да имплементираме коректно обхождането на хеш-таблицата, реализирахме интерфейса **Iterable<DictionaryEntry<K, V>>**, който има метод, връщащ итератор по елементите на хеш-таблицата. За да реализираме метода итератора, първо прехвърляме всички елементи в ArrayList, а след

това връщаме неговия итератор. Следва пример как можем да използваме нашата реализация на хеш-таблица и нейният итератор:

```
public class HashDictionaryExample {
   public static void main(String[] args) {
      HashDictionary<String, Integer> marks =
         new HashDictionary<String, Integer>();
      marks.put("Pepi", 3);
      marks.put("Kiro", 4);
      marks.put("Mimi", 6);
      marks.put("Pepi", 5); // replace key "Pepi"
      marks.remove("Kiro"); // remove key "Kiro"
      marks.remove("123"); // key not found

      // Use the iterator to traverse all entries
      for (DictionaryEntry<String, Integer> entry : marks) {
         System.out.print(entry + " ");
      }
      // Output: [Mimi, 6] [Pepi, 5]
   }
}
```

В примерната имплементация на хеш-таблица има още една особеност. Методът **findChain()** не е реализиран напълно коректно, но проблемът трудно може да се прояви. Наистина в повечето случаи тази реализация ще работи без проблем. Но какво ще стане, ако добавяме елементи до безкрай? В един прекрасен момент, когато капацитетът е станал 2^{31} и се наложи да го разширим, то при умножение на това число с 2 ще получим -2 (вж. секцията за представяне на отрицателни числа в главата "Бройни системи"). След това при опит за създаване на нов масив с размер -2 естествено ще бъде хвърлено изключение и изпълнението на метода ще бъде прекратено.

За да напълните тази реализация на хеш-таблица с толкова много двойки (ключ, стойност) ви е необходима доста RAM памет. При зададена 1024MB памет на виртуалната машина изключението **OutOfMemmoryError** се хвърля още преди да са добавени 6 милиона и 300 хиляди двойки от тип **Integer** на ключа и стойността. На практика не е добра идея да се работи с изключително много данни на веднъж. Ето защо не трябва да ви притеснява и фактът, че максималната стойност за брой на елементи на масиви и колекции в Java е 2 147 483 647.

За да зададете максималната памет, която да заеме ваша програма преди получаването на изключителната ситуация **OutOfMemmoryError**, трябва при стартиране на виртуалната машина да подадете параметъра **-XmxSIZE**, където **SIZE** е обемът памет, който искате да зададете. По подразбиране тази стойност е само 64 MB. Например: ако искате да стартирате вашата програма с най-много 512 MB оперативна памет, трябва да изпълните:

```
java -Xmx512m SomeClassWithMainMethod
```

Методи за решаване на колизиите от тип отворена адресация (open addressing)

Нека сега разгледаме методите за разрешаване на колизиите, алтернативни на нареждането в списък. Най-общо идеята при тях е, че в случай на колизия се опитваме да сложим новата двойка на някоя свободна позиция от таблицата. Методите се различават по това как се избира къде да се търси свободно място за новата двойка. Освен това трябва да е възможно и намирането на тази двойка на новото ѝ място.

Основен недостатък на този тип методи спрямо нареждането в списък е, че са неефективни при голяма степен на запълненост (близка до 1).

Линейно пробване (linear probing)

Този метод е един от най-лесните за имплементация. Линейното пробване най-общо представлява следният простичък код:

```
int newPosition = (oldPosition + i) % capacity;
```

Тук **capacity** е капацитетът на таблицата, **oldPostion** е позицията, за която получаваме колизия, а **i** е номер на поредното пробване. Ако ново-получената позиция е свободна, то мястото се използва за новодобавената двойка, в противен случай пробваме отново, като увеличаваме **i** с единица. Възможно е пробването да е както напред така и назад. Пробване назад става като вместо да прибавяме, вадим **i** от позицията, в която имаме колизия.

Предимство на този метод е сравнително бързото намиране на нова позиция. За нещастие има изключително висока вероятност, ако на едно място е имало колизия, след време да има и още. Това на практика води до силна неефективност.

Използването на линейно пробване като метод за решаване на проблема с колизиите е неефективно и трябва да се избягва.

Квадратично пробване (Quadratic probing)

Това е класически метод за решаване на проблема с колизиите. Той се различава от линейното пробване с това, че за намирането на нова позиция се използва квадратна функция на **i** (номер на поредно пробване). Ето как би изглеждало едно такова решение:

```
int newPosition = (oldPosition + c1*i + c2*i*i) % capacity;
```

Тук се появяват две константи **c1** и **c2**. Иска се **c2** да е различна от 0, защото в противен случай се връщаме на линейно пробване.

От избора на **c1** и **c2** зависи на кои позиции спрямо началната ще пробваме. Например, ако **c1** и **c2** са равни на 1, ще пробваме последователно **oldPosition**, **oldPosition + 2**, **oldPosition + 6**, …. За таблица с капацитет от вида **2n**, е най-добре да се изберат **c1** и **c2** равни на **0.5**.

Квадратичното пробване е по-ефективно от линейното.

Двойно хеширане (double hashing)

Както става ясно и от името на този метод, при повторното хеширане за намиране на нова позиция се прави повторно хеширане на получения хеш-код, но с друга хеш-функция, съвсем различна от първата. Този метод е по-добър от линейното и квадратичното пробване, тъй като всяко следващо пробване зависи от стойността на ключа, а не от позицията определена за ключа в таблицата. Това има смисъл, защото позицията за даден ключ зависи от текущия капацитет на таблицата.

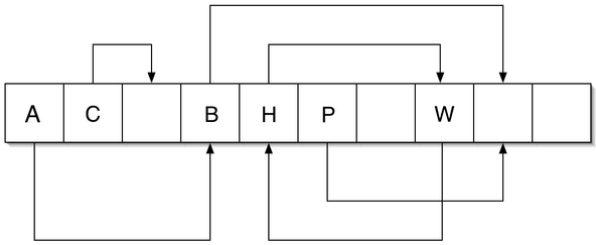
Кукувиче хеширане (cuckoo hashing)

Кукувичето хеширане е сравнително нов метод с отворена адресация за справяне с колизиите. Той е бил представен за пръв път от R. Pagh и F. Rodler през 2001 година. Името му идва от поведението, наблюдавано при някои видове кукувици. Майките кукувици избутват яйца и/или малките на други птици извън гнездото им, за да оставят техните яйца там и така други птици да се грижат за техните яйца (и малки след излюпването).

Основната идея на този метод е да се използват две хеш-функции вместо една. По този начин ще разполагаме не с една, а с две позиции, на които можем да поставим елемент в речника. Ако единият от двата елемента е свободен, то просто слагаме елемента на свободна позиция. Ако пък и двете позиции са заети, то слагаме новият елемент на една от двете позиции, като той "изритва" елемента, който до сега се е намирал там. На свой ред "изритания" елемент отива на своята алтернативна позиция, като "изритва" някой друг елемент, ако е необходимо. Новият "изритан" повтаря процедурата и така, докато не се достигне свободна позиция или докато не се получи зацикляне. Във втория случай цялата таблица се построява наново с по-голям размер и с нови хеш-функции.

На картинката по-долу е показана примерна схема на хеш-таблица, която използва кукувиче хеширане. Всяка клетка, която съдържа елемент има връзка към алтернативната клетка за ключа, който се намира в нея. Сега ще проиграем различни ситуации за добавяне на нов елемент.

Ако поне една от двете хеш-функции ни даде свободна клетка, то няма проблем. Слагаме елемента в една от двете. Нека обаче и двете хеш функции са дали заети клетки и на случаен принцип сме избрали една от тях.



Нека също предположим, че това е клетката, в която се намира A. Новият елемент изритва A от неговото място, A на свой ред отива на алтернативната си позиция и изритва B, от неговото място. Алтернативното място за B обаче е свободно, така че добавянето завършва успешно.

Да предположим, че клетката, от която се опитва да изрита елемент, новият елемент е тази, в която се намира H. Тогава се получава зацикляне тъй като H и W образуват цикъл. В този случай трябва да се изпълни пресъздаване на таблицата, използвайки нови хеш-функции и по-голям размер.

В най-опростената си версия този метод има константен достъп до елементите си и то в най-лошия случай, но това е изпълнено само при ограниченото, че фактора на запълване е по-малък от 0.5.

Използването на три различни хеш-функции, вместо две може да доведе до ефективна горна граница на фактора на запълване до над 0.9.

Проучвания показват, че кукувичето хеширане и неговите варианти могат да бъдат много по-ефективни от широко използваните днес нареждане в списък и методите с отворено адресиране. Въпреки това все още този метод остава широко неизвестен и неизползван в практиката.

Структура от данни "множество"

В тази секция ще разгледаме абстрактната структура от данни множество (set) и две нейни типични реализации. Ще обясним предимствата и недостатъците им и в какви ситуации коя от имплементациите да предпочитаме.

Абстрактна структура данни "множество"

Множествата са колекции, в които няма повтарящи се елементи. В контекста на Java това ще означава, че за всеки обект от множества извиквайки метода му **`equals()`**, като подаваме като аргумент някои от другите обекти във множеството резултатът винаги ще е **`false`**.

Някои множества позволяват присъствието в себе си и на **`null`**, други не.

Освен, че не допуска повтарящи се обекти, друго важно нещо, което отличава множеството от списъците и масивите е, че неговите елементи си нямат номер. Елементите на множеството не могат да бъдат достъпвани по някакъв друг ключ, както е при речниците. Самите елементи играят ролята на ключ.

Единственият начин да достъпите обект от множество е като разполагате със самия обект или евентуално с обект, който е еквивалентен на него. Затова на практика достъпваме всички елементи на дадено множество наведнъж, докато го обхождаме в цикъл. Например чрез разширената конструкцията за **for** цикъл.

Основните операции, които се дефинират от структурата множество са следните:

- **boolean add(element)** – добавя в множеството зададен елемент, като ако вече има такъв елемент, връща **false**, а в противен случай **true**.
- **boolean contains(element)** – проверява дали множеството съдържа посочения елемент. Ако го има връща **true**, а в противен случай **false**.
- **boolean remove(element)** – премахва посочения елемент от множеството, ако съществува. Връща дали е елементът е бил намерен.
- **Set intersect(Set other)** – връща сечението на две множества – множество, което съдържа всички елементи, които са едновременно и в едното и в другото множество.
- **Set union(Set other)** – връща обединението на две множества – множество, което съдържа всички елементи, които са или в едното или в другото множество или и в двете.
- **boolean containsAll(Set other)** – проверява дали дадено множество е подмножество на текущото. Връща **true** при положителен отговор и **false** при отрицателен.

В Java имаме основен интерфейс, който описва структурата от данни множество. Това е интерфейсът **java.util.Set**. Той има две основни имплементации и те са чрез хеш-таблица (**HashSet**) и чрез червено-черно дърво (**TreeSet**). Ако разгледаме внимателно имплементацията на тези класове ще видим, че те всъщност представляват речници, при които елементът е едновременно ключ и стойност за наредената двойка. Естествено, когато е удобно да работим с множества, трябва да ги предпочитаме, пред това да използваме речник.

Операции обединение и сечение на множества

Повечето от описаните по-горе методи ги има декларирани и в интерфейса **Set**. Някои от операциите, обаче нямат стандартна имплементация и се реализират малко по-специфично.

За да реализираме операцията union (обединение), трябва сами да напишем кода, реализиращ такава функционалност, примерно чрез използване на метода **AddAll()**:

```
public static <E> Set<E> union(Set<E> set1, Set<E> set2) {
   // Here we use HashSet but you can use TreeSet if appropriate
```

```
    Set<E> union = new HashSet<E>();
    union.addAll(set1);
    union.addAll(set2);
    return union;
}
```

Забележете, че създаваме нов обект за множеството, съдържащо обединението, а не добавяме първото множество към второто. Тук използваме описаната в следващият параграф имплементация **HashSet**. Благодарение на това след изпълнението на кода първото и второто множество продължават да съдържат точно елементите, които са съдържали и преди това.

Друга операция, която не ни е дадена наготово, е сечението на множества (intersection). За да я реализираме можем да използваме метода **retainAll()**, който премахва всички елементи от дадено множество, които не се съдържат в друго подадено като параметър. Ето една реализация на сечение, отново използваща **HashSet**:

```
public static <E> Set<E> intersect(Set<E> set1, Set<E> set2) {
    // Here we use HashSet but you can use TreeSet if appropriate
    Set<E> intersect = new HashSet<E>();
    intersect.addAll(set1);
    intersect.retainAll(set2);
    return intersect;
}
```

Отново създаваме нов обект за резултата. Добавяме в резултата всички елементи от първото множество и след това премахваме от резултата всички елементи, които не се съдържат във второто множество.

Реализация с хеш-таблица – клас HashSet<T>

Реализацията на множество с хеш-таблица в Java е класът **HashSet<T>**. Този клас подобно на **HashMap<K, V>** има конструктори, в които може да се зададат степен на запълване и начален капацитет. Те имат същият смисъл, защото тук отново използваме хеш-таблица. Винаги е добре, ако знаете предварително приблизително размерът на множеството, да го задавате изрично.

За обединението може да използвате следната оценка на максималния брой елементи в резултата:

```
set1.size() + set2.size()
```

За сечението оценката на максималния брой елементи в резултата е:

```
Math.min(set1.size(), set2.size())
```

Ето един изключително прост пример, който демонстрира използване на множества и описаните в предния параграф методи за обединение и сечение:

```
Set<String> javaStudents = new HashSet<String>();
javaStudents.add("S. Nakov");
javaStudents.add("V. Kolev");
javaStudents.add("V. Tsanev");
Set<String> linuxStudents = new HashSet<String>();
linuxStudents.add("D. Alexiev");
linuxStudents.add("V. Tsanev");

System.out.println("Java Students: " + javaStudents);
System.out.println("Linux Students: " + linuxStudents);
System.out.println("Java or Linux Students: " +
  union(javaStudents, linuxStudents));
System.out.println("Java and Linux Students: " +
  intersect(javaStudents, linuxStudents));
```

Резултатът от изпълнението е:

```
Java Students: [V. Tsanev, S. Nakov, V. Kolev]
Linux Students: [D. Alexiev, V. Tsanev]
Java or Linux Students: [D. Alexiev, V. Tsanev, S. Nakov, V. Kolev]
Java and Linux Students: [V. Tsanev]
```

Обърнете внимание, че **V. Tsanev** присъства и в двете множества, но в обединението се появява само веднъж. Именно това показва че един елемент може да се съдържа най-много веднъж в дадено множество.

Реализация с черно-червено дърво – клас TreeSet<T>

Класът **TreeSet<T>** представлява множество, реализирано чрез червено-черно дърво. То има свойството, че в него елементите се пазят подредени по големина. Това е причината в него да можем да добавяме само елементи които са сравними. Припомняме, че в Java това обикновено означава, че обектите са от клас, който имплементира **Comparable<T>**. Ако това не е така, тук също можем да използваме интерфейса **Comparator<T>**, чрез който да задаваме наредба или да подменяме естествената.

Ще демонстрираме работата с класа **TreeSet<T>** с един не толкова формален и скучен пример. Нека имаме софтуерна компания, която искала всички нейни служители да се чувстват възможно най-добре по време на работния ден. Една от задачите, които си е поставило ръководството, била да се събере списък с всички любими на служителите музикални групи. Целта била да се състави списък с групите, подредени по азбучен ред. Всеки ден случайно се избирала една буква и звучали песни на групите започващи с

тази буква. За простота всички имена написани от фирмените служители били на латиница. След като получили данните за всеки от служителите и уволнили хората с музикални вкусове противоречащи на фирмената политика, се получил списък с не много групи. Проблемът бил, че имало много повторения. Нашата цел е от даден неподреден по никакъв начин списък с групи да премахнем повторенията и да оставим само различните групи, като ги изведем по азбучен ред. Използването на **TreeSet** не е единственият начин да се реши тази задача, но тук ще демонстрираме колко просто става това с негова помощ:

```
String[] bandNames = new String[] {
      "manowar", "blind guardian", "dio",
      "grave digger", "slayer", "seputltura", "kiss", "sodom",
      "manowar", "megadeth", "dio", "judas priest", "slayer",
      "manowar", "kreator", "blind guardian", "iron maiden",
      "accept", "seputltura", "iced earth", "manowar", "slayer",
      "manowar", "helloween", "running wild", "manowar",
      "sodom", "kiss", "iron maiden", "manowar", "manowar",
      "sodom", "manowar", "slayer", "blind guardian", "accept",
      "grave digger", "accept", "seputltura", "dio",
      "running wild", "manowar", "iron maiden", "kiss",
      "manowar", "manowar", "kiss", "manowar", "slayer",
      "seputltura", "manowar", "manowar", "blind guardian",
      "iron maiden", "sodom", "dio", "accept", "manowar",
      "slayer", "megadeth", "dio", "manowar", "running wild",
      "grave digger", "accept", "kiss", "manowar", "iron maiden",
      "manowar", "judas priest", "sodom", "iced earth",
      "manowar", "dio", "iron maiden", "manowar", "slayer",
      "manowar" };

SortedSet<String> uniqueBandNames = new TreeSet<String>();
for (String bandName : bandNames) {
   uniqueBandNames.add(bandName);
}

System.out.println("List of sorted and unique band names:");
for (String bandName : uniqueBandNames) {
   System.out.println(bandName);
}
```

След изпълнението на програмата получаваме списъка:

```
List of sorted and unique band names:
accept
blind guardian
dio
grave digger
helloween
```

```
iced earth
iron maiden
judas priest
kiss
kreator
manowar
megadeth
running wild
seputltura
slayer
sodom
```

Фирмата била изненадана от резултата, но пък тъй като за щастие всяка от групите е била достатъчно продуктивна през годините на нейното съществуване, лесно можели да заменят случайната буква със случайна група от списъка.

В крайна сметка важно е да си дадете сметка, че работата с множества е наистина лесна и проста. Ако познавате добре тяхната структура, ще можете и да ги ползвате ефективно и на място.

Упражнения

1. Напишете програма, която премахва всички числа, които се срещат нечетен брой пъти в дадена редица. Например, ако имаме началната редица {4, 2, 2, 5, 2, 3, 2, 3, 1, 5, 2, 6, 6, 6}, трябва да я редуцираме до редицата {5, 3, 3, 5}.
2. Реализирайте клас **DictHashSet<T>**, базиран на класа **HashDictionary <K, V>**, който разгледахме по-горе.
3. Реализирайте хеш-таблица, която съхранява тройки стойности (ключ1, ключ2, стойност) и позволява бързо търсене по двойка ключове и добавяне на тройки стойности.
4. Реализирайте хеш-таблица, която позволява по даден ключ да съхраняваме повече от една стойност.
5. Реализирайте хеш-таблица, която използва кукувиче хеширане с 3 хеш функции за разрешаване на колизиите.
6. Дадени са три редици от числа, дефинирани чрез формулите:
   - $f_1(0) = 1$; $f_1(k) = 2*f_1(k-1) + 3$; $f_1 = \{1, 5, 13, 29, \ldots\}$
   - $f_2(0) = 2$; $f_2(k) = 3*f_2(k-1) + 1$; $f_2 = \{2, 7, 22, 67, \ldots\}$
   - $f_3(0) = 2$; $f_3(k) = 2*f_3(k-1) - 1$; $f_3 = \{2, 3, 5, 9, \ldots\}$

   Напишете програма, която намира сечението и обединението на множествата от членовете на редиците в интервала [0; 100000]: $f_1 * f_2$; $f_1 * f_3$; f_2

* f_3; $f_1 * f_2 * f_3$; $f_1 + f_2$; $f_1 + f_3$; $f_2 + f_3$; $f_1 + f_2 + f_3$. Със символите + и * означаваме съответно обединение и сечение на множества.

7. * Дефинирайте клас **`TreeMultiSet<T>`**, който позволява да пазим съвкупност от елементи, подредени по големина и позволява повторения на някои от елементите. Реализирайте операциите добавяне на елемент, търсене на броя срещания на даден елемент, изтриване на елемент, итератор, намиране на най-малък / най-голям елемент, изтриване на най-малък / най-голям елемент. Реализирайте възможност за подаване на външен **`Comparator<T>`** за сравнение на елементите.

8. * Даден е списък с времената на пристигане и заминаване на всички автобуси от дадена автогара. Да се напише програма, която използвайки **`TreeSet`** и **`HashSet`** класовете по даден интервал (начало, край) намира броя автобуси, които успяват да пристигнат и да напуснат автогарата. Пример:

   Имаме данните за следните автобуси: [08:24-08:33], [08:20-09:00], [08:32-08:37], [09:00-09:15]. Даден е интервалът [08:22-09:05]. Броят автобуси, които идват и си тръгват в рамките на този интервал е 2.

9. * Дадена е редица **`P`** с цели числа ($1 < P < 50\,000$) и число N. Щастлива подредица в редицата **`P`** наричаме всяка съвкупност, състояща се от последователни числа от **`P`**, чиято сума е N. Да си представим, че имаме редицата **`S`**, състояща се от всички щастливи подредици в **`P`**, подредени в намаляващ ред спрямо дължината им. Напишете програма, която извежда първите 10 елемента на **`S`**. Пример:

   Имаме N=5 и редицата **`P={1, 1, 2, 1, -1, 2, 3, -1, 1, 2, 3, 5, 1, -1, 2, 3}`**.

   Редицата **`S`** се състои от следните 13 подредици на **`P`**:

   - **[1, -1, 2, 3, -1, 1]**
   - **[1, 2, 1, -1, 2]**
   - **[1, -1, 2, 3]**
   - **[2, 3, -1, 1]**
   - **[3, -1, 1, 2]**
   - **[-1, 1, 2, 3]**
   - **[1, -1, 2, 3]**
   - **[1, 1, 2, 1]**
   - **[5, 1, -1]**
   - **[2, 3]**
   - [2, 3]
   - [2, 3]

- [5]

Първите 10 елемента на P са дадени с удебелен шрифт.

Решения и упътвания

1. Използвайте **HashMap** и **ArrayList**.
2. Използвайте за ключ и за стойност една и съща стойност – елементът от множеството.
3. Използвайте хеш-таблица от хеш-таблици.
4. Ползвайте **HashMap<K, ArrayList<V>>**.
5. Можете за първа хеш-функция да ползвате **hashCode() % size**, за втора да ползвате **(hashCode() * 31 + 7) % size**, а за трета – (**hashCode() * hashCode() + 19) % size**).
6. Намерете всички членове на трите редици в посочения интервал и след това използвайки **HashSet<Integer>** реализирайте обединение и сечение на множества, след което направете исканите пресмятания.
7. Класът **TreeMultiSet<T>** можете да реализираме чрез **TreeMap<K, Integer>**, който пази броя срещания на всеки от ключовете.
8. Очевидното решение е да проверим всеки от автобусите дали пристига и си тръгва в посочения интервал. Според условието на задачата, обаче, трябва да ползваме класовете **TreeSet** и **HashSet**.

   Решението е такова: можем да дефинираме клас **TimeInterval** и да си направим две множества **TreeSet<TimeInterval>**, в които да пазим разписанията на автобусите, подредени съответно по час на пристигане и по час на отпътуване. Ще трябва да дефинираме и две имплементации на **Comparator<TimeInterval>**. Накрая можем да намерим множествата на всички автобуси, които пристигат след началния час и на всички автобуси, отпътуващи преди крайния час. Сечението на тези множества дава търсените автобуси. Сечението можем да намерим с **HashSet<TimeInterval>** при подходящо дефинирани **hashCode()** и **equals()**.
9. Първата идея за решаване на задачата е проста: с два вложени цикъла намираме всички щастливи подредици на редицата P, след което ги сортираме по дължината им и накрая извеждаме първите 10. Това, обаче няма да работи добре, ако броят щастливи подредици са десетки милиони.

   Ще опишем една идея за по-ефективно решение. Ще използваме класа **TreeMultiSet<T>**. В него ще съхраняваме първите 10 подредици от **S**, т.е. мултимножество от щастливите подредици на **P**, подредени по дължина в намаляващ ред. Когато имаме 10 подредици в мултимножеството и добавим нова 11-та подредица, тя ще застане на мястото си заради компаратора, който сме дефинирали. След това можем веднага да

изтрием последната подредица от мултимножеството, защото тя не е сред първите 10. Така във всеки един момент ще пазим текущите 10 най-дълги подредици. По този начин ще консумираме много по-малко памет и ще избегнем сортирането накрая. Имплементацията няма да е лесна, така че отделете достатъчно време!

Глава 19. Структури от данни – съпоставка и препоръки

Автор

Светлин Наков

Николай Недялков

В тази тема...

В настоящата тема ще съпоставим една с друга структурите данни, които разгледахме до момента, по отношение на скоростта, с която извършват основните операции (добавяне, търсене, изтриване и т.н.). Ще дадем конкретни препоръки в какви ситуации какви структури от данни да ползваме. Ще обясним кога да предпочетем хеш-таблица, кога масив, кога динамичен масив, кога множество, реализирано чрез хеш-таблица и кога балансирано дърво. Всички тези структури имат вградена в Java платформата имплементация. От нас се иска единствено да можем да преценяваме кога коя структура да ползваме, за да пишем ефективен и надежден програмен код. Именно на това е посветена настоящата тема – на ефективната работа със структури от данни.

Защо са толкова важни структурите данни?

Може би се чудите защо отделяме толкова голямо внимание на структурите данни и защо ги разглеждаме в такива големи детайли? Причината е, че сме си поставили за задача да ви направим мислещи софтуерни инженери. Без да познавате добре основните структури от данни в програмирането и основните компютърни алгоритми, вие не можете да бъдете добри програмисти и рискувате да си останете обикновени "занаятчии". Който владее добре структурите от данни и алгоритми и успее да си развие мисленето в посока правилното им използване, има големи шансове да стане добър софтуерен инженер – който анализира проблемите в дълбочина и предлага ефективни решения.

По темата защо са важни структурите от данни и алгоритмите има изписани стотици книги. Особено впечатляващи са четирите тома на Доналд Кнут, озаглавени "The Art of Computer Programming", в които структурите от

данни и алгоритмите са разгледани в над 2500 страници. Един автор дори е озаглавил книга с отговора на въпроса "защо структурите от данни са толкова важни". Това е книгата на Никлаус Вирт "Алгоритми + структури от данни = програми", в която се разглеждат отново структурите данни и фундаменталните алгоритми в програмирането.

Структурите от данни и алгоритмите стоят в основата на програмирането. За да станете добри програмисти, е необходимо да познавате основните структури от данни и алгоритми и да се научите да ги прилагате по подходящ начин.

В много голяма степен и нашата книга е насочена именно към изучаването на основните структури от данни и алгоритми в програмирането, като сме се стремили да ги илюстрираме в контекста на съвременното софтуерно инженерство с Java платформата.

Сложност на алгоритъм

Не може да се говори за ефективност на алгоритми и структури от данни, без да се използва понятието "сложност на алгоритъм", с което вече се сблъскахме няколко пъти под една или друга форма. Няма да даваме математическа дефиниция, за да не натоварваме читателите, а ще дадем неформално обяснение.

Сложност на алгоритъм е метрика, която отразява порядъка на броя операции, необходими за изпълнение на дадена операция или алгоритъм като функция на обема на входните данни. Формулирано още по-просто, сложност е груба, приблизителна оценка на броя стъпки за изпълнение на даден алгоритъм. Означава се най-често с нотацията O(f), където f е функция на обема на входните данни.

Сложността може да бъде константна, логаритмична, линейна, n*log(n), квадратична, кубична, експоненциална и друга. Това означава, че се изпълняват съответно константен, логаритмичен и т.н. брой стъпки за решаването на даден проблем.

Сложност на алгоритъм е груба оценка на броя стъпки, които алгоритъмът ще направи в зависимост от обема на входните данни.

Типични сложности на алгоритмите

Ще обясним какво означават видовете сложност чрез следната таблица:

Сложност	Означение	Описание
константна	O(1)	За извършване на дадена операция са необходими константен брой стъпки (примерно 1, 5, 10 или друго число) и този брой не зависи от обема на входните данни.
логаритмична	O(log(N))	За извършване на дадена операция върху N елемента са необходими брой стъпки от порядъка на log(N), където основата на логаритъма е най-често 2. Примерно алгоритъм със сложност O(log(N)) за N = 1 000 000 ще направи около 20 стъпки (с точност до константа).
линейна	O(N)	За извършване на дадена операция върху N елемента са необходими приблизително толкова стъпки, колкото са елементите. Примерно за 1 000 елемента са нужни около 1 000 стъпки. Броят елементи и броят операции са линейно зависими, примерно броят стъпки е около N/2 или 3*N за N елемента.
	O(n*log(n))	За извършване на дадена операция върху N елемента са необходими приблизително N*log(N) стъпки. Примерно при 1 000 елемента са нужни около 10 000 стъпки.
квадратична	$O(n^2)$	За извършване на дадена операция са необходими N^2 на брой стъпки, където N характеризира обема на входните данни. Примерно за дадена операция върху 100 елемента са необходими 10 000 стъпки. Ако броят стъпки е в квадратна зависимост спрямо обема на входните данни, то сложността е квадратична.
кубична	$O(n^3)$	За извърхване на дадена операция са необходими от порадъка на N^3 стъпки, където N характеризира обема на входните данни. Примерно при 100

		елемента се изпълняват около 1 000 000 стъпки.
експоненциална	$O(2^n)$, $O(N!)$, $O(n^k)$, …	За извърпване на дадена операция или изчисление са необходими брой стъпки, който е в експоненциалназависимост спрямо размера на входните данни. Например при N=10 експоненциалната функция 2^N има стойност 1024, при N=20 има стойност 1 048 576, а при N=100 функцията има стойност, която е число с около 30 цифри.

При оценката на сложност константите не се взимат предвид, тъй като не влияят съществено на броя операции. По тази причина алгоритъм, който извършва N стъпки и алгоритми, които извършват съответно N/2 и 3*N стъпки се считат за линейни и за приблизително еднакво ефективни.

Сложност и време за изпълнение

Скоростта на изпълнение на програмата е в пряка зависимост от сложността на алгоритъма, който се изпълнява. Ако тази сложност е малка, програмата ще работи бързо, дори за голям брой елементи. Ако сложността е голяма, програмата ще работи бавно или въобще няма да работи (ще заспи) при голям брой елементи.

Ако вземем един средностатистически компютър от 2008 година, можем да приемем, че той изпълнява около 50 000 000 елементарни операции в секунда. Разбира се, това число трябва да ви служи единствено за груб ориентир. Различните процесори работят с различна скорост и различните елементарни операции се изпълняват с различна скорост. Все пак, за да имаме някакъв ориентир, можем да направим следните изводи за скоростта на изпълнение на дадена програма в зависимост от сложността на алгоритъма и обема на входните данни:

алгоритъм	**10**	**20**	**50**	**100**	**1 000**	**10 000**	**100 000**
O(1)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(log(n))	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(n)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(n*log(n))	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.

$O(n^2)$	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	2 сек.	3-4 мин.
$O(n^3)$	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	20 сек.	5.55 часа	231.5 дни
$O(2^n)$	< 1 сек.	< 1 сек.	260 дни	зас-пива	зас-пива	заспива	заспива
O(n!)	< 1 сек.	зас-пива	зас-пива	зас-пива	зас-пива	заспива	заспива
$O(n^n)$	3-4 мин.	зас-пива	зас-пива	зас-пива	зас-пива	заспива	заспива

От таблицата можем да направим много изводи:

- Алгоритми с константна, логаритмична и линейна сложност са толкова бързи, че не можем да усетим забавяне, дори при относително голям вход.
- Сложността O(n*log(n)) е близка до линейната и също работи толкова, бързо, че трудно можем да усетим забавяне.
- Квадратични алгоритми работят добре до няколко хиляди елемента.
- Кубични алгоритми работят добре при под 1 000 елемента.
- Като цяло т.нар. полиномиални алгоритми (тези, които не са експоненциални) се считат за бързи и работят добре за хиляди елементи.
- Експоненциалните алгоритми като цяло не работят и трябва да ги избягваме (ако е възможно). Ако имаме експоненциално решение за дадена задача, може да се каже, че нямаме решение, защото то ще работи само ако елементите са под 10-20. Съвременната криптография разчита точно на това, че не са известни бързи (неекспоненциални) алгоритми за откриване на тайните ключове, които се използват за шифриране на данните.

Ако решите една задача с експоненциална сложност, това означава, че сте я решили само за много малък размер на входните данни и в общия случай решението ви не работи.

Разбира се, данните в таблицата са само ориентировъчни. Понякога може да се случи линеен алгоритъм да работи по-бавно от квадратичен или квадратичен да работи по-добре от O(n*log(n)). Причините за това могат да са много:

- Възможно е константите за алгоритъм с малка сложност да са големи и това да прави алгоритъма бавен като цяло. Например, ако имаме алгоритъм, който прави 50*n стъпки и друг, който прави 1/100*n*n

стъпки, то за стойности до 5000 квадратичният алгоритъм е по-бърз от линейния.

- Понеже оценката на сложността се прави за най-лошия случай, е възможно квадратичен алгоритъм да работи по-добре от алгоритъм O(n*log(n)) в 99% от случаите. Можем да дадем пример с алгоритъма QuickSort (бързо сортиране), който в средния случай работи малко по-добре от MergeSort (сортиране чрез сливане), но в най-лошия случай QuickSort прави от порядъка на n^2 стъпки.
- Възможно е алгоритъм, който е оценен, че работи с линейна сложност, да не работи толкова бързо, колкото се очаква заради неточна оценка на сложността. Например, ако търсим дадена дума в масив от думи, сложността е линейна, но на всяка стъпка се извършва сравнение на символни низове, което не е елементарна операция и може да отнеме много повече от една стъпка.

Сложност по няколко променливи

Сложността може да зависи и от няколко входни променливи едновременно. Примерно, ако търсим елемент в правоъгълна матрица с размери M на N, то скоростта на търсенето зависи и от M и от N. Понеже в най-лошия случай трябва да обходим цялата матрица, то ще направим най-много M*N на брой стъпки. Така сложността се оценява като O(M*N).

Най-добър, най-лош и среден случай

Сложността на алгоритмите се оценява обикновено в най-лошия случай (при най-неблагоприятния сценарий). Това означава, че в средния случай те могат да работят и по-бързо, но в най-лошия случай работят с посочената сложност и не по-бавно.

Да вземем един пример: търсене на елемент в масив по даден ключ. За да намерим търсения ключ, трябва да проверим в най-лошия случай всички елементи на масива. В най-добрия случай ще имаме късмет и ще намерим търсения ключ още в първия елемент. В средния случай можем да очакваме да проверим средно половината елементи на масива докато намерим търсения. Следователно в най-лошия случай сложността е O(N), т.е. линейна, в средния случай сложността е O(N/2) = O(N), т.е. линейна (при оценяване на сложност константите се пренебрегват). В най-добрия случай имаме константна сложност O(1), защото изпълняваме само една стъпка и откриваме търсения елемент.

Приблизително оценена сложност

Понякога е трудно да оценим точно сложността на даден алгоритъм, тъй като изпълняваме операции, за които не знаем колко бързо се изпълняват. Да вземем за пример търсенето на дадена дума в масив от текстове. Трябва да обходим масива и във всеки от текстовете да търсим със **`substring()`** или

с регулярен израз дадената дума. Ако имаме 10 000 текста, това бързо ли ще работи? Не може да се каже, защото трябва да знаем колко са обемни текстовете. Можем да оценим сложността на най-малко O(L), където L е сумата от дължините на всички текстове. При някои специални ситуации, обаче търсенето зависи съществено и от дължината на търсената дума и тази оценка може да се окаже силно занижена.

Сложност по памет

Освен броя стъпки чрез функция на входните данни могат да се измерват и други ресурси, които алгоритъма използва, например памет, брой дискови операции и т.н. За някои алгоритми скоростта на изпълнение не е толкова важна, колкото обема на паметта, която ползват. Например, ако един алгоритъм използва оперативна памет от порядъка на N^2, той вероятно ще страда от недостиг на памет при N=100 000 (тогава ще му трябват от порядъка на 9 GB оперативна памет).

Оценяване на сложност – примери

Ще дадем няколко примера, с които ще ви покажем как можете да оценявате сложността на вашите алгоритми и да преценявате дали ще работи бързо написаният от вас програмен код:

Ако имаме единичен цикъл от 1 до N, сложността му е линейна – O(N):

```
int findMaxElement(int[] array) {
   int max = array[0];
   for (int i=0; i<array.length; i++) {
      if (array[i] > max)
         max = array[i];
   }
   return max;
}
```

Този код ще работи добре, дори при голям брой елементи.

Ако имаме два вложени цикъла от 1 до N, сложността им е квадратична – $O(N^2)$. Пример:

```
int findInversions(int[] array) {
   int inversions = 0;
   for (int i=0; i<array.length; i++)
      for (int j = i+1; j<array.length; j++)
         if (array[i] > array[j])
            inversions++;
   return inversions;
}
```

Този код ще работи добре, ако елементите не са повече от няколко хиляди или десетки хиляди.

Ако имаме три вложени цикъла от 1 до N, сложността им е кубична – O(N^3). Пример:

```
long sum3(int n) {
   long sum = 0;
   for (int a = 0; a < n; a++)
      for (int b = 0; b < n; b++)
         for (int c = 0; c < n; c++)
            sum += a * b * c;
   return sum;
}
```

Този код ще работи добре, ако елементите в масива са под 1 000.

Ако имаме два вложени цикъла съответно от 1 до N и от 1 до M, сложността им е квадратична – O(N*M). Пример:

```
long sumMN(int n, int m) {
   long sum = 0;
   for (int x = 0; x < n; x++)
      for (int y = 0; y < m; y++)
         sum += x * y;
   return sum;
}
```

Скоростта на този код зависи от две променливи. Кодът ще работи добре, ако M, N < 10 000 или ако поне едната променлива има малка стойност.

Не винаги три вложени цикъла означават кубична сложност. Ето един пример, при който сложността е O(N*M):

```
long sumMN(int n, int m) {
   long sum = 0;
   for (int x = 0; x < n; x++)
      for (int y = 0; y < m; y++)
         if (x == y)
            for (int i = 0; i < n; i++)
               sum += i * x * y;
   return sum;
}
```

Най-вътрешния цикъл се изпълнява точно min(M, N) пъти и не оказва съществено влияние върху скоростта на алгоритъма. Горният код изпълнява приблизително N*M + min(M,N)*N стъпки.

При използване на рекурсия сложността е по-трудно да се определи. Ето един пример:

```
long factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}
```

В този пример сложността е линейна – O(N), защото функцията **factorial()** се изпълнява точно веднъж за всяко от числата 1, 2, ..., n.

Ето една рекурсивна функция, за която е много трудно да се сметне сложността:

```
long fibonacci(int n) {
   if (n == 0)
      return 1;
   else if (n == 1)
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Функцията се извиква толкова пъти, колкото е числото на Фибоначи с номер n+1. Можем грубо да оценим сложността и по друг начин: понеже на всяка стъпка от изпълнението на функцията се извършват средно по 2 рекурсивни извиквания, то броят рекурсивни извиквания би трябвало да е от порядъка на 2^n, т.е. имаме експоненциална сложност.

Същата функция за изчисление на n-тото число на Фибоначи можем да напишем с линейна сложност по следния начин:

```
public static long fibonacci(int n) {
   long fn = 1;
   long fn_1 = 1;
   long fn_2 = 1;
   for (int i = 2; i < n; i++) {
      fn = fn_1 + fn_2;
      fn_2 = fn_1;
      fn_1 = fn;
   }
   return fn;
}
```

Виждате, че оценката на сложността ни помага да предвидим, че даден код ще работи бавно, още при да сме го изпълнили и ни подсказва, че трябва да търсим по-ефективно решение.

Сравнение на основните структури от данни

След като се запознахме с понятието сложност на алгоритъм, вече сме готови да направим съпоставка на основните структури от данни, които разгледахме до момента и да оценим за какво време всяка от тях извършва основните операции като добавяне, търсене и други:

структура	добавяне	търсене	изтриване	достъп по индекс
`Array`	O(N)	O(N)	O(N)	O(1)
`LinkedList`	O(1)	O(N)	O(N)	O(N)
`ArrayList`	O(1)	O(N)	O(N)	O(1)
`Stack`	O(1)	-	O(1)	-
`Queue`	O(1)	-	O(1)	-
`HashMap`	O(1)	O(1)	O(1)	-
`TreeMap`	O(log(N))	O(log(N))	O(log(N))	-
`HashSet`	O(1)	O(1)	O(1)	-
`TreeSet`	O(log(N))	O(log(N))	O(log(N))	-

Кога да използваме дадена структура?

Нека разгледаме всяка от посочените в таблицата структури от данни поотделно и обясним в какви ситуации е подходящо да се ползва такава структура и как се получават сложностите, дадени в таблицата.

Масив (Array)

Масивите са наредени съвкупности от N елемента, до които достъпът става по индекс. Масивите представляват област от паметта с фиксиран размер. Добавянето на нов елемент в масив е много бавна операция, защото реално трябва да се задели нов масив с размерност по-голяма с 1 от текущата и да се прехвърлят старите данни в новия масив. Търсенето в масив изисква сравнение на всеки елемент с търсената стойност. В средния случай са необходими N/2 сравнения. Изтриването от масив е много бавна операция, защото е свързана със заделяне на масив с размер с 1 по-малък от текущия и преместване на всички елементи без изтрития в новия масив. Достъпът по индекс става директно.

Масивите трябва да се ползват само когато трябва да обработим фиксиран брой елементи, до които е необходим достъп по индекс. Например, ако сортираме числа, можем да запишем числата в масив и да приложим някой от добре известните алгоритми за сортиране.

Ползвайте масиви, когато трябва да обработите фиксиран брой елементи, до които ви трябва достъп по индекс.

Свързан / двусвързан списък (LinkedList)

Свързаният списък и неговият вариант двусвързан списък съхраняват подредена съвкупност от елементи. Добавянето е бърза операция, но по-бавна от добавяне в **`ArrayList`**, защото всяко добавяне заделя памет, което работи със скорост, която трудно може да бъде предвидена. Търсенето е бавна операция, защото е свързано с обхождане на всички елементи. Достъпът до елемент по индекс е бавна операция, защото в свързания списък няма индексиране и се налага обхождане на списъка. Изтриването на елемент по индекс е бавна операция, защото достигането до елемента с посочения индекс е бавно. Изтриването по стойност на елемент също е бавно, защото включва в себе си търсене.

Свързаният списък може бързо (с константна сложност) да добавя и изтрива елементи от двата си края, поради което е удобен за имплементация на стекове, опашки и други подобни структури.

Свързан списък в практиката се използва много рядко, защото динамичният масив (**`ArrayList`**) изпълнява почти всички операции, които могат да бъдат изпълнени с **`LinkedList`**, но за повечето от тях работи по-бързо.

Ползвайте **`ArrayList`**, когато ви трябва свързан списък – той работи не по-бавно, а ви дава по-голяма бързина и удобство. Ползвайте **`LinkedList`**, ако имате нужда от добавяне и изтриване на елементи в двата края на структурата.

Ползвайте свързан списък, когато трябва да добавяте и изтривате елементи от двата края на списъка.

Динамичен масив (ArrayList)

Динамичният масив (**`ArrayList`**) е една от най-използваните в практиката структура от данни. Той няма фиксиран размер, както масивите и има директен достъп по индекс, за разлика от свързания списък.

Динамичният масив вътрешно съхранява елементите си в масив, който е по-голям от броя съхранени елементи. При добавяне в масива обикновено има свободно място и това отнема константно време. Понякога масивът се препълва и трябва да се разшири. Това отнема линейно време, но се случва много рядко. В крайна сметка усреднената сложност на добавянето на елемент към **`ArrayList`** е константна – O(1). Тази усреднена сложност се нарича **амортизирана сложност**. Амортизирана сложност означава, че ако добавим последователно 10 000 елемента, ще извършим общо брой

стъпки от порядъка на 10 000, въпреки че някои от тях ще се изпълнят за константно време, а други – за линейно.

Търсенето в **ArrayList** е бавна операция, защото трябва да се обходят всички елементи. Изтриването по индекс или по стойност се изпълнява за линейно време. Изтриването е бавна операция, защото е свързана с преместване на всички елементи, които са след изтрития с една позиция наляво. Достъпът по индекс в **ArrayList** става непосредствено, за константно време, тъй като елементите се съхраняват вътрешно в масив.

В крайна сметка **ArrayList** комбинира добрите страни на масивите и на списъците, заради което е предпочитана структура данни в много ситуации. Например, ако трябва да обработим текстов файл и да извлечем от него всички думи, отговарящи на даден регулярен израз, най-удобната структура, в която можем да ги натрупваме, е **ArrayList**.

Динамичният масив (**ArrayList**) е подходящ, когато трябва често да добавяме елементи и искаме да запазваме реда им на добавяне и да ги достъпваме често по индекс. Ако често търсим или изтриваме елемент, **ArrayList** не е подходяща структура.

Ползвайте `ArrayList`**, когато трябва бързо да добавяте елементи и да ги достъпвате по индекс.**

Стек (Stack)

Стекът е структура от данни, в която са дефинирани 3 операции: добавяне на елемент на върха на стека, изтриване на елемент от върха на стека и извличане на елемент от върха на стека, без да го изтриваме. Всички тези операции се изпълняват бързо, с константна сложност. Операциите търсене и достъп по индекс не се поддържат.

Стекът е структура с поведение LIFO (last in, first out) – последен влязъл, пръв излязъл. Използва се, когато трябва да моделираме такова поведение, например, ако трябва да пазим пътя до текущата позиция при рекурсивно търсене.

Ползвайте стек, когато е необходимо да реализирате поведението "последен влязъл, пръв излязъл" (LIFO).

Опашка (Queue)

Опашката е структура от данни, в която са дефинирани две операции: добавяне на елемент и извличане на елемента, който е наред. Тези две операции се изпълняват бързо, с константна сложност, тъй като опашката обикновено се имплементира чрез свързан списък. Припомняме, че свързаният списък може да добавя и изтрива бързо елементи в двата си края.

Поведението на структурата опашка е FIFO (first in, first out) – пръв влязъл, пръв излязъл. Операциите търсене и достъп по индекс не се поддържат. Опашката по естествен начин моделира списък от чакащи хора, задачи или други обекти, които трябва да бъдат обработени последователно, в реда на постъпването им.

Като пример за използване на опашка можем да посочим реализацията на алгоритъма "търсене в ширина", при който се започва от даден начален елемент и неговите съседи се добавят в опашка, след което се обработват по реда им на постъпване и по време на обработката им техните съседи се добавят в опашката. Това се повтаря докато не се стигне до даден елемент, до който търсим път.

Ползвайте опашка, когато е необходимо да реализирате поведението "пръв влязъл, пръв излязъл" (FIFO).

Речник, реализиран с хеш-таблица (HashMap)

Структурата "речник" предполага съхраняване на двойки ключ-стойност като осигурява бързо търсене по ключ. При реализацията с хеш-таблица (класа **`HashMap`** в Java) добавянето, търсенето и изтриването на елементи работят много бързо – със средна сложност константа. Операцията достъп по индекс не е достъпна, защото елементите в хеш-таблицата се нареждат по почти случаен начин и редът им на постъпване не се запазва.

`HashMap` съхранява вътрешно елементите си в масив, като поставя всеки елемент на позиция, която се дава от хеш-функцията. По този начин масивът се запълва частично – в някои клетки има стойност, докато други стоят празни. Ако трябва да се поставят няколко стойности в една и съща клетка, те се нареждат в свързан списък (колизиите се решават чрез chaining). Когато степента на запълненост на хеш-таблицата надвиши 75%, тя нараства двойно и всички елементи заемат нови позиции. Тази операция е с линейна сложност, но се изпълнява толкова рядко, че амортизираната сложност на операцията добавяне си остава константа.

Хеш-таблицата има една особеност: При неблагоприятно избрана хеш-функция основните операции могат да работят доста неефективно и да се достигне линейна сложност. В практиката, обаче, това почти не се случва. Затова се счита, че хеш-таблицата е най-бързата структура от данни, която осигурява добавяне и търсене по ключ.

Хеш-таблицата предполага, че всеки ключ се среща в нея най-много веднъж. Ако добавим последователно два елемента с един и същ ключ, последният постъпил ще измести предходния и в крайна сметка ще изгубим един елемент. Това е важна особеност, с която трябва да се съобразяваме. Ако се налага в един ключ да съхраняваме няколко стойности, можем да ползваме **`ArrayList`** като стойност за всеки ключ.

Хеш-таблица се препоръчва винаги, когато ни трябва бързо търсене по ключ. Например, ако трябва да преброим колко пъти се среща в текстов файл всяка дума измежду дадено множество думи, можем да ползваме `HashMap<string, int>` като ползваме за ключ търсените думи, а за стойност – колко пъти се срещат във файла.

Ползвайте хеш-таблица, когато искате бързо да добавяте елементи и да търсите по ключ.

Много програмисти (най-вече начинаещите) живеят със заблудата, че основното предимство на хеш-таблицата е в удобството да търсим дадена стойност по нейния ключ. Всъщност основното предимство въобще не е това. Търсене по ключ можем да реализираме и с масив и със списък и дори със стек. Няма проблем, всеки може да ги реализира. Можете да си дефинирате клас `Entry`, който съхранява ключ и стойност и да си работите с масив или списък от Entry елементи. Можете да си реализирате търсене, но при всички положения то ще работи бавно. Това е големият проблем при списъците и масивите – нямат бързо търсене. За разлика от тях хеш-таблицата може да търси бързо и да добавя бързо нови елементи.

Основното предимство на хеш-таблицата пред останалите структури от данни е изключително бързото търсене и добавяне на елементи, а не толкова удобството на работа!

Речник, реализиран с дърво (TreeMap)

Реализацията на структурата от данни "речник" чрез червено-черно дърво (класът `TreeMap`) е структура, която предполага съхранение на двойки ключ-стойност, при което ключовете са подредени по големина. Структурата осигурява бързо изпълнение на основните операции (добавяне на елемент, търсене по ключ и изтриване на елемент). Сложността, с която се изпълняват тези операции, е логаритмична – O(log(N)). Това означава 10 стъпки при 1000 елемента и 20 стъпки при 1 000 000 елемента.

За разлика от хеш-таблиците, където при лоша хеш-функция може да се достигне до линейна сложност на търсенето и добавянето, при `TreeMap` броят стъпки за основните операции в средния и в най-лошия случай е един и същ – $\log_2(N)$.

Отново, както при хеш-таблиците, един ключ може да се среща в структурата най-много веднъж. Ако искаме да поставяме няколко стойности под един и същ ключ, трябва да ползваме за стойност някакъв списък, примерно `ArrayList`.

`TreeMap` държи вътрешно елементите си в червено-черно дърво, подредени по ключа. Това означава, че ако обходим структурата (чрез нейния итератор), ще получим елементите сортирани в нарастващ ред по ключа им. Понякога това може да е много полезно.

Използвайте **TreeMap** в случаите, в които ви трябва структура, в която бързо да добавяте, бързо да търсите и имате нужда от извличане на елементите, сортирани в нарастващ ред. В общия случай **HashMap** работи малко по-бързо от **TreeMap** и е за предпочитане.

Като пример за използване на **TreeMap** можем да дадем следната задача: Да се намерят всички думи в текстов файл, които се срещат точно 10 пъти и да се отпечатат по азбучен ред. Това е задача, която можете да решите също така успешно и с **HashMap**, но ще ви се наложи да направите едно сортиране повече.

Ползвайте TreeMap**, когато искате бързо да добавяте елементи и да търсите по ключ и елементите ще ви трябват сортирани по ключ.**

Множество, реализирано с хеш-таблица (HashSet)

Структурата от данни "множество" представлява съвкупност от елементи, сред които няма повтарящи се. Основните операции са добавяне на елемент към множеството, проверка за принадлежност на елемент към множеството (търсене) и премахване на елемент от множеството (изтриване). Операцията търсене по индекс не се поддържа, т.е. нямаме директен достъп до елементите.

Множество, реализирано чрез хеш-таблица (класът **HashSet**), е частен случай на хеш-таблица, при който имаме само ключове, а стойностите, записани под всеки ключ са без значение.

Както и при хеш-таблицата, основните операции в структурата от данни **HashSet** са реализирани с константна сложност O(1). Както и при хеш-таблицата, при неблагоприятна хеш-функция може да се стигне до линейна сложност на основните операции, но в практиката това почти не се случва.

Като пример за използването на **HashSet** можем да посочим задачата за намиране на всички различни думи в даден текстов файл.

Ползвайте HashSet**, когато трябва бързо да добавяте елементи към множество и да проверявате дали даден елемент е от множеството.**

Множество, реализирано с дърво (TreeSet)

Множество, реализирано чрез червено-черно дърво (класът **TreeSet**), е частен случай на **TreeMap**, в който ключовете и стойностите съвпадат.

Както и при **TreeMap** структурата, основните операции в **TreeSet** са реализирани с логаритмична сложност O(log(N)), като тази сложност е една и съща и в средния и в най-лошия случай.

Като пример за използването на **TreeSet** можем да посочим задачата за намиране на всички различни думи в даден текстов файл и отпечатването им по азбучен ред.

Ползвайте TreeSet**, когато трябва бързо да добавяте елементи към множество и да проверявате дали даден елемент е от множеството и освен това елементите ще ви трябват сортирани в нарастващ ред.**

Други структури в Java платформата

Ако разгледаме стандартните библиотеки на Java платформата, ще се убедим, че освен разгледаните до момента колекции (имплементации на **List**, **Map** и **Set**), тя ни предоставя и много техни варианти със специално предназначение. Такива са например неизменимите (read only) колекции (**Collections.UnmodifiableCollection**) и синхронизираните колекции, които позволяват конкурентен достъп от няколко нишки (**Collections.SynchronizedCollection**, **Vector**, **Hashtable**, **ConcurrentHashMap**) и др. Те се ползват в много специфични ситуации, така е малко вероятно да имате нужда от тях.

Измежду структурите от стандартната библиотека на Java, които не сме споменавали до момента полезни могат да ви бъдат **PriorityQueue** и **LinkedHashSet**.

Приоритетна опашка (PriorityQueue)

Абстрактната структура от данни "приоритетната опашка" прилича на опашка, но елементите, които попадат в нея постъпват с даден приоритет. Елементите напускат опашката по ред, определен от приоритета им. Елементите с по-висок приоритет напускат опашката по рано, отколкото елементите с по-нисък приоритет. Елементите с еднакъв приоритет напускат опашката в неопределен ред.

Класът **PriorityQueue** имплементира ефективно приоритетна опашка чрез дървовидна структура, наречена **двоична пирамида** (binary heap). Тази имплементация гарантира добавяне и премахване на елементи със сложност O(log(N)) и начално построяване на структурата пирамида от неподреден списък с линейна сложност. За разлика от класа **TreeSet** в **PriorityQueue** можем да имаме повтарящи се елементи.

Ще демонстрираме как можем да използваме класа **PriorityQueue** с един пример:

```
import java.util.PriorityQueue;

public class PriorityQueueExample {

```

```
   static class Person implements Comparable<Person> {
      String name;
      int priority;

      public Person(String name, int priority) {
         this.name = name;
         this.priority = priority;
      }

      @Override
      public int compareTo(Person p) {
         if (this.priority > p.priority) {
            return 1;
         } else if (this.priority < p.priority) {
            return -1;
         } else {
            return 0;
         }
      }

      @Override
      public String toString() {
         return "[" + name + " : " + priority + "]";
      }
   }

   public static void main(String[] args) {
      PriorityQueue<Person> queue = new PriorityQueue<Person>();
      queue.add(new Person("Maria", 8));
      queue.add(new Person("Peter", 5));
      queue.add(new Person("George", 3));
      while (! queue.isEmpty()) {
         System.out.println(queue.poll());
      }
      // Output is sorted according to the priority:
      // [George : 3]
      // [Peter : 5]
      // [Maria : 8]
   }
}
```

В примера създаваме клас **Person**, който дефинира име и приоритет. Имплементираме интерфейса **Comparable**, за да дефинираме наредба по полето **priority**. Предефинираме **toString()** метода, за да можем свободно да отпечатваме **Person** обекти в конзолата. След това създаваме приоритетна опашка и добавяме в нея три обекта от тип **Person** в случаен ред. След това ги изваждаме и отпечатваме. От резултата се убеждаваме,

че приоритетната опашка изважда елементите по реда, дефиниран от приоритета им.

Множество, запазващо наредбата (LinkedHashSet)

Класът **LinkedHashSet** представлява множество, имплементирано с хеш-таблица, което запазва реда на постъпване на елементите. **LinkedHashSet** е комбинация между **HashSet** и двусвързан списък. При обхождане елементите се обработват по реда им на постъпване, който се запазва в допълнителен списък. Класът е удобен, когато искаме да работим с множество с голяма бързина (**HashSet**), но не искаме да губим реда на постъпване на елементите.

Избор на структура от данни – примери

Сега ще дадем няколко задачи, при които изборът на подходяща структура от данни е от решаващо значение за ефективността на тяхното решение. Целта е да ви покажем типични ситуации, в които се използват разгледаните структури от данни и да ви научим в какви ситуации какви структури от данни да ползвате.

Генериране на подмножества

Дадено е множество от символни низове S, примерно S = {море, бира, пари, кеф}. Да се напише програма, която отпечатва всички подмножества на S.

Задачата има много и различни по идея решения, но ние ще се спрем на следното решение: Започваме от празно подмножество (с 0 елемента):

{}

Към него добавяме всеки от елементите на S и получаваме съвкупност от подмножества с по 1 елемент:

{море}, {бира}, {пари}, {кеф}

Към всяко от получените едноелементни подмножества добавяме всеки от елементите на S, който не се съдържа в съответното подмножество и получаваме всички двуелементни подмножества:

{море, бира}, {море, пари}, {море, кеф}, {бира, пари}, {бира, кеф}, {пари, кеф}

Ако продължим по същия начин, ще получим всички 3-елементни подмножества и след тях 4-елементните т. н. до N-елементните подмножества.

Как да реализираме този алгоритъм? Трябва да изберем подходящи структури от данни, нали?

Можем да започнем с избора на структурата, която съхранява началното множество от елементи S. Тя може да е масив, свързан списък, динамичен масив (**ArrayList**) или множество, реализирано като **TreeSet** или **HashSet**. За

да си отговорим на въпроса коя структура е най-подходяща, нека помислим кои са операциите, които ще трябва да извършваме върху тази структура. Сещаме се само за една операция – обхождане на всички елементи на S. Тази операция може да бъде реализирана ефективно с всяка от изброените структури. Избираме масив, защото е най-простата структура от възможните и с него се работи най-лесно.

Следва да изберем структурата, в която ще съхраняваме едно от подмножествата, които генерираме, примерно {море, кеф}. Отново си задаваме въпроса какви са операциите, които извършваме върху такова подмножество от думи. Операциите са проверка за съществуване на елемент и добавяне на елемент, нали? Коя структура реализира бързо тази двойка операции? Масивите и списъците не търсят бързо, речниците съхраняват двойки ключ-стойност, което не е нашия случай. Остана да видим структурата множество. Тя поддържа бързо търсене и бързо добавяне. Коя имплементация да изберем – **TreeSet** или **HashSet**? Нямаме изискване за сортиране на думите по азбучен ред, така че избираме по-бързата имплементация – **HashSet**.

Остана да изберем още една структура от данни – структурата, в която съхраняваме съвкупност от подмножества от думи, примерно:

{море, бира}, {море, пари}, {море, кеф}, {бира, пари}, {бира, кеф}, {пари, кеф}

В тази структура трябва да можем да добавяме, както и да обхождаме елементите ѝ последователно. На тези изисквания отговарят структурите списък, стек, опашка и множество. Във всяка от тях можем да добавяме бързо и да обхождаме елементите ѝ. Ако разгледаме внимателно алгоритъма за генериране на подмножествата, ще забележим, че всяко от тях се обработва в стил "пръв генериран, пръв обработен". Подмножеството, което първо е било получено, първо се обработва и от него се получават подмножествата с 1 елемент повече, нали? Следователно на нашия алгоритъм най-точно ще пасне структурата от данни опашка. Можем да опишем алгоритъма така:

1. Започваме от опашка, съдържаща празното множество {}.
2. Взимаме поредния елемент **subset** от опашката и към него се опитваме да добавим всеки елемент от S, който не се съдържа в **subset**. Резултатът е множество, което добавяме към опашката.
3. Повтаряме последната стъпка докато опашката свърши.

Виждате, че с разсъждения стигнахме до класическия алгоритъм "търсене в ширина". След като знаем какви структури от данни да използваме, имплементацията става бързо и лесно. Ето как би могла да изглежда тя:

```
String[] words = {"море", "бира", "пари", "кеф"};
Queue<Set<String>> subsetsQueue = new LinkedList<Set<String>>();
```

```
Set<String> emptySet = new HashSet<String>();
subsetsQueue.offer(emptySet);
while (! subsetsQueue.isEmpty()) {
   Set<String> subset = subsetsQueue.poll();
   System.out.println(subset);
   for (String element : words) {
      if (! subset.contains(element)) {
         Set<String> newSubset = new HashSet<String>();
         newSubset.addAll(subset);
         newSubset.add(element);
         subsetsQueue.offer(newSubset);
      }
   }
}
```

Ако изпълним горния код, ще се убедим, че той генерира успешно всички подмножества на S, но някои от тях ги генерира по няколко пъти. Изглежда не сме се сетили за повторенията. Как можем да ги избегнем?

Да номерираме думите по техните индекси:

море → 0
бира → 1
пари → 2
кеф → 3

Понеже подмножествата {1, 2, 3} и {2, 1, 3} са всъщност едно и също подмножество, за да нямаме повторения, ще наложим изискването да генерираме само подмножества, в които индексите са подредени по големина. Можем вместо множества от думи да пазим множества от индекси, нали? В тези множества от индекси ни трябват две операции: добавяне на индекс и взимане на най-големия индекс, за да добавяме само индекси, по-големи от него. Очевидно **HashSet** вече не ни върши работа, но можем успешно да ползваме **ArrayList**, в който елементите са наредени по големина и най-големия елемент по естествен начин е последен в списъка.

В крайна сметка нашия алгоритъм добива следната форма:

1. Нека N е броя елементи в S. Започваме от опашка, съдържаща празния списък {}.
2. Взимаме поредния елемент **subset** от опашката. Нека **start** е най-големия индекс в **subset**. Към **subset** добавяме всички индекси, който са по-големи от **start** и по-малки от N. В резултат получаваме няколко нови подмножества, които добавяме към опашката.
3. Повтаряме последната стъпка докато опашката свърши.

Ето как изглежда реализацията на новия алгоритъм:

```
import java.util.*;

public class Subsets {

   private static String[] words =
      {"море", "бира", "пари", "кеф"};

   public static void main(String[] args) {
      Queue<ArrayList<Integer>> subsetsQueue =
         new LinkedList<ArrayList<Integer>>();
      ArrayList<Integer> emptySet = new ArrayList<Integer>();
      subsetsQueue.offer(emptySet);
      while (! subsetsQueue.isEmpty()) {
         ArrayList<Integer> subset = subsetsQueue.poll();
         print(subset);
         int start = -1;
         if (subset.size() > 0) {
            start = subset.get(subset.size()-1);
         }
         for (int index = start+1; index < words.length; index++){
            ArrayList<Integer> newSubset =
               new ArrayList<Integer>();
            newSubset.addAll(subset);
            newSubset.add(index);
            subsetsQueue.offer(newSubset);
         }
      }
   }

   private static void print(ArrayList<Integer> subset) {
      System.out.print("[ ");
      for (int i=0; i<subset.size(); i++) {
         int index = subset.get(i);
         System.out.print(words[index] + " ");
      }
      System.out.println("]");
   }

}
```

Ако изпълним програмата, ще получим очаквания резултат:

```
[ ]
[ море ]
[ бира ]
[ пари ]
[ кеф ]
```

```
[ море бира ]
[ море пари ]
[ море кеф ]
[ бира пари ]
[ бира кеф ]
[ пари кеф ]
[ море бира пари ]
[ море бира кеф ]
[ море пари кеф ]
[ бира пари кеф ]
[ море бира пари кеф ]
```

Подреждане на студенти

Даден е текстов файл, съдържащ данните за група студенти и курсовете, които те изучават. Файлът изглежда по следния начин:

```
Кирил  | Иванов    | Java
Милена | Стефанова | PHP
Благой | Иванов    | Java
Петър  | Иванов    | Linux
Стефка | Василева  | C++
Милена | Василева  | Java
```

Да се напише програма, която отпечатва всички курсове и за всеки от тях студентите, които са ги записали, подредени първо по фамилия, след това по име (ако фамилиите съвпадат).

Задачата можем да реализираме чрез хеш-таблица, която по име на курс пази списък от студенти. Избираме хеш-таблица, защото в нея можем бързо да търсим по име на курс.

За да изпълним условието за подредба по фамилия и име, при отпечатването на студентите от всеки курс ще трябва да сортираме съответния списък. Другият вариант е да ползваме **`TreeSet`** за студентите от всеки курс (понеже той вътрешно е сортиран), но понеже може да има студенти с еднакви имена, трябва да ползваме **`TreeSet<List<String>>`**. Става сложно. Избираме по-лесния вариант – да ползваме **`ArrayList<Student>`** и да го сортираме преди да го отпечатаме.

При всички случаи ще трябва да реализираме интерфейса **`Comparable`**, за да дефинираме наредбата на елементите от тип **`Student`** според условието на задачата. Необходимо е първо да сравняваме фамилията и при еднаква фамилия да сравняваме името. Нека дефинираме класа Student и имплементираме **`Comparable<Student>`**. Получаваме нещо такова:

```
public class Student implements Comparable<Student> {
```

```
   private String firstName;
   private String lastName;

   public Student(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public String getName() {
      return this.firstName + " " + this.lastName;
   }

   public int compareTo(Student student) {
      int result = this.lastName.compareTo(student.lastName);
      if (result == 0) {
         result = this.firstName.compareTo(student.firstName);
      }
      return result;
   }

   @Override
   public String toString() {
      return firstName + " " + lastName;
   }
}
```

Сега вече можем да напишем кода, който прочита студентите и техните курсове и ги записва в хеш-таблица, която по име на курс пази списък със студентите в този курс (**HashMap<String, ArrayList<Student>>**). След това вече е лесно – итерираме по курсовете, сортираме студентите и ги отпечатваме:

```
// Read the file and build the hash-table of courses
HashMap<String, ArrayList<Student>> courses =
   new HashMap<String, ArrayList<Student>>();
Scanner input =
   new Scanner(new File("Students.txt"), "windows-1251");
try {
   while (input.hasNext()) {
      String line = input.nextLine();
      String[] studentEntry = line.split("\\s*\\|\\s*");
      String firstName = studentEntry[0];
      String lastName = studentEntry[1];
      String course = studentEntry[2];

      ArrayList<Student> students = courses.get(course);
      if (students == null) {
```

```
            // New course -> create a list of students for it
            students = new ArrayList<Student>();
            courses.put(course, students);
         }
         Student student = new Student(firstName, lastName);
         students.add(student);
      }
   } finally {
      input.close();
   }

   // Print the courses and their students
   Set<String> coursesNames = courses.keySet();
   for (String course : coursesNames) {
      System.out.println("Course " + course + ":");
      ArrayList<Student> students = courses.get(course);
      Student[] studentsArr =
         students.toArray(new Student[students.size()]);
      Arrays.sort(studentsArr);
      for (Student student : studentsArr) {
         System.out.printf("\t%s\n", student);
      }
   }
```

Примерният код чете студентите от файла **Students.txt** и парсва редовете му по регулярния израз "**\s*\|\s***", т. е. ползва за разделител всички вертикални черти, евентуално предхождани и следвани от незадължително празно пространство. След прочитането на всеки студент се проверява хеш-таблицата дали съдържа неговия курс. Ако курсът е намерен, студентът се добавя към списъка със студенти за този курс. Ако курсът не е намерен, се създава нов списък, към него се добавя студента и списъкът се записва в хеш-таблицата под ключ името на курса.

Отпечатването на курсовете и студентите е просто. От хеш-таблицата се извличат всички ключове. Това са имената на курсовете. За всеки курс се извлича списък от студентите му. Списъкът се превръща в масив, за да може да се сортира с **Arrays.sort()**. Сортирането се извършва, както е дефинирано в компаратора на класа **Student** – първо по фамилия, а при еднакви фамилии – по име. Накрая сортираните студенти се отпечатват чрез предефинирания в тях виртуален метод **toString()**. Ето как изглежда изходът от горната програма:

```
Course Linux:
   Петър Иванов
Course PHP:
   Милена Стефанова
Course C++:
```

```
   Стефка Василева
Course Java:
   Милена Василева
   Благой Иванов
   Кирил Иванов
```

Подреждане на телефонен указател

Даден е текстов файл, който съдържа имена на хора, техните градове и телефони. Файлът изглежда по следния начин:

```
Киро | Варна    | 052 / 23 45 67
Пешо | София    | 02 / 234 56 78
Мими | Пловдив | 0888 / 22 33 44
Лили | София    | 0899 / 11 22 33
Дани | Варна    | 0897 / 44 55 66
```

Да се напише програма, която отпечатва всички градове по азбучен ред и за всеки от тях отпечатва всички имена на хора по азбучен ред и съответния им телефон.

Задачата можем да решим по много начини, например като сортираме по два критерия: на първо място по град и на второ място по телефон и след това отпечатаме телефонния указател.

Нека, обаче решим задачата без сортиране, като използваме стандартните структури от данни в Java. Искаме да имаме в сортиран вид градовете. Това означава, че трябва да ползваме структура, която държи елементите си в сортиран вид. Такава е балансираното дърво – **TreeSet** или **TreeMap**. Понеже всеки запис от телефонния указател съдържа освен град и други данни, е по-удобно да имаме **TreeMap**, който по ключ име на град пази списък от хора и техните телефони. Понеже искаме списъкът на хората за всеки град да е сортиран по имената на хората, можем отново да ползваме **TreeMap**. Като ключ можем да държим име на човек, а като стойност – неговият телефон. В крайна сметка получаваме структурата **TreeMap<String, TreeMap<String, String>>**. Следва примерна имплементация, която показва как можем да решим задачата с тази структура:

```
// Read the file and build the phone book
TreeMap<String, TreeMap<String, String>> phonesByTown =
   new TreeMap<String, TreeMap<String, String>>();
Scanner input = new Scanner(
   new File("PhoneBook.txt"), "windows-1251");
try {
   while (input.hasNext()) {
      String line = input.nextLine();
      String[] phoneBookEntry = line.split("\\s*\\|\\s*");
```

```
      String name = phoneBookEntry[0];
      String town = phoneBookEntry[1];
      String phone = phoneBookEntry[2];

      TreeMap<String, String> phoneBook = phonesByTown.get(town);
      if (phoneBook == null) {
         // This town is new. Create a phone book for it
         phoneBook = new TreeMap<String, String>();
         phonesByTown.put(town, phoneBook);
      }
      phoneBook.put(name, phone);
   }
} finally {
   input.close();
}

// Print the phone book by towns
Set<String> towns = phonesByTown.keySet();
for (String town : towns) {
   System.out.println("Town " + town + ":");
   TreeMap<String, String> phoneBook = phonesByTown.get(town);
   for (Map.Entry<String, String> entry : phoneBook.entrySet()) {
      String name = entry.getKey();
      String phone = entry.getValue();
      System.out.printf("\t%s - %s\n", name, phone);
   }
}
```

Ако изпълним този код с вход примерния телефонен указател, ще получим очаквания резултат:

```
Town Варна:
   Дани - 0897 / 44 55 66
   Киро - 052 / 23 45 67
Town Пловдив:
   Мими - 0888 / 22 33 44
Town София:
   Лили - 0899 / 11 22 33
   Пешо - 02 / 234 56 78
```

Търсене в телефонен указател

Даден е телефонен указател, записан в текстов файл, който съдържа имена на хора, техните градове и телефони. Имената на хората могат да бъдат във формат малко име или прякор или име + фамилия или име + презиме + фамилия. Файлът има следния вид:

```
Киро Киров         | Варна   | 052 / 23 45 67
Мундьо             | София   | 02 / 234 56 78
Киро Киров Иванов  | Пловдив | 0888 / 22 33 44
Лили Иванова       | София   | 0899 / 11 22 33
Киро               | Плевен  | 064 / 88 77 66
Киро бирата        | Варна   | 0897 / 44 55 66
Киро               | Плевен  | 0897 / 44 55 66
```

Възможно е да има няколко души, записани под едно и също име, дори и от един и същ град. Възможно е някой да има няколко телефона и в такъв случай той се изписва няколко пъти във входния файл. Телефонният указател може да бъде доста голям (до 1 000 000 записа).

Даден е файл със заявки за търсене. Заявките са два вида:

- Търсене по име / прякор / презиме / фамилия. Заявката има вида **`list(name)`**.
- Търсене по име / прякор / презиме / фамилия + град. Заявката има вида **`find(name, town)`**.

Ето примерен файл със заявки:

```
list(Киро)
find(Пешо, София)
list(Лили)
list(Киров)
find(Иванов, Пловдив)
list(Баба)
```

Задачата е по даден телефонен указател и файл със заявки да се върнат всички отговори на заявките за търсене. За всяка заявка да се изведе списък от записите в телефонния указател, които й съответстват или "Not found", ако заявката не намира нищо. Заявките могат да са голям брой (примерно 50 000).

Тази задача не е толкова лесна, колкото предходните. Едно лесно за реализация решение е при всяка заявка да се сканира целият телефонен указател и да се изваждат всички записи, в които има съвпадения с търсената информация. Това, обаче ще работи бавно, защото записите могат да са много и заявките могат да са много. Необходимо е да намерим начин да търсим бързо, без да сканираме всеки път целия телефонен указател.

В хартиените телефонни указатели телефоните са дадени по имената на хората, подредени в азбучен ред. Сортирането няма да ни помогне, защото някой може да търси по име, друг по фамилия, а трети – по прякор и име на град. Ние трябва да можем да търсим по всичко това. Въпросът е как да го направим?

Ако поразсъждаваме малко, ще се убедим, че в задачата се изисква търсене по всяка от думите, които се срещат в първата колона на телефонния указател и евентуално по комбинацията дума от първата колона и град от втората колона. Знаем, че най-бързото търсене, което можем да реализираме, се прави с хеш-таблица. Въпросът е какво да използваме за ключ и какво да използваме за стойност в хеш-таблицата.

Дали пък да не ползваме няколко хеш-таблици: една за търсене по първата дума от първата колона, още една за търсене по втората колона, една за търсене по град и т.н. Ако се замислим още малко, ще си зададем въпроса – защо са ми няколко хеш-таблици? Не може ли да търсим само в една хеш-таблица. Ако имаме "Петър Иванов", в таблицата ще сложим под ключ "Петър" неговия телефон и същевременно под ключ "Иванов" същия телефон. Ако някой търси една от двете думи, ще намери телефона на Петър.

До тук добре, обаче как ще търсим по име и по град, примерно "Петър от Варна"? Възможно е първо да намерим всички с име "Петър" и от тях да отпечатаме само тези, които са от Варна. Това ще работи, но ако има много хора с име Петър, търсенето по град ще е бавно. Тогава защо не направим хеш-таблица по ключ име на човек и стойност друга хеш-таблица, която по град връща списък от телефони? Това би трябвало да работи. Нещо подобно правихме в предходната задача, нали?

Хрумва ни нещо още по-умно. Не може ли в главната хеш-таблица за телефонния указател да сложим под ключ "Петър от Варна" телефоните на всички, които се казват Петър и са от Варна? Изглежда това ще реши проблема и ще можем да използваме само една хеш-таблица за всички търсения.

В крайна сметка стигаме до следния алгоритъм: Четем ред по ред телефонния указател и за всяка дума от името на човека $\mathbf{d_1}$, $\mathbf{d_2}$, ..., $\mathbf{d_k}$ и за всеки град **t** добавяме текущия запис от указателя под следните ключове: $\mathbf{d_1}$, $\mathbf{d_2}$, ..., $\mathbf{d_k}$, "$\mathbf{d_1}$ **от** **t**", "$\mathbf{d_2}$ **от** **t**", …, "$\mathbf{d_k}$ **от** **t**". За да можем да търсим без значение на регистъра (главни или малки букви), прави предварително всички букви малки. След това търсенето е тривиално – просто търсим в хеш-таблицата подадената дума или ако ни подадат дума **d** и град **t**, търсим по ключ "**d от t**". Понеже за един и същ ключ може да има много телефони, ползваме за стойност списък от символни низове (**ArrayList<String>**). Нека разгледаме една имплементация на описания алгоритъм:

```
import java.io.*;
import java.util.*;

public class PhoneBookFinder {

   private static final String PHONE_BOOK_FILE = "PhoneBook.txt";
   private static final String QUEIRES_FILE = "Queries.txt";
```

```
public static void main(String[] args) throws IOException {
   HashMap<String, ArrayList<String>> phoneBook =
      readPhoneBook(PHONE_BOOK_FILE);
   processQueries(QUEIRES_FILE, phoneBook);
}

private static HashMap<String, ArrayList<String>>
readPhoneBook(String fileName) throws FileNotFoundException {
   HashMap<String, ArrayList<String>> phoneBook =
      new HashMap<String, ArrayList<String>>();
   Scanner input =
      new Scanner(new File(fileName), "windows-1251");
   try {
      while (input.hasNext()) {
         String entry = input.nextLine();
         String[] phoneBookEntry =
            entry.split("\\s*\\|\\s*");
         String names = phoneBookEntry[0];
         String town = phoneBookEntry[1];
         String[] nameTokens = names.split("\\s+");
         for (String name : nameTokens) {
            addToPhoneBook(phoneBook, name, entry);
            String nameAndTown =
               combineNameAndTown(town, name);
            addToPhoneBook(phoneBook, nameAndTown, entry);
         }
      }
   } finally {
      input.close();
   }
   return phoneBook;
}

private static String combineNameAndTown(
      String town, String name) {
   return name + " от " + town;
}

private static void addToPhoneBook(
      HashMap<String, ArrayList<String>> phoneBook,
      String name, String entry) {
   name = name.toLowerCase();
   ArrayList<String> entries = phoneBook.get(name);
   if (entries == null) {
      entries = new ArrayList<String>();
      phoneBook.put(name, entries);
   }
```

```
      entries.add(entry);
   }

   private static void processQueries(String fileName,
         HashMap<String, ArrayList<String>> phoneBook)
         throws IOException {
      Scanner input =
         new Scanner(new File(fileName), "windows-1251");
      try {
         while (input.hasNext()) {
            String query = input.nextLine();
            processQuery(phoneBook, query);
         }
      } finally {
         input.close();
      }
   }

   private static void processQuery(HashMap<String,
         ArrayList<String>> phoneBook, String query) {
      if (query.startsWith("list(")) {
         String name = query.substring(
            "list(".length(), query.length()-1);
         name = name.trim().toLowerCase();
         printAllMatches(name, phoneBook);
      } else if (query.startsWith("find(")) {
         int commaIndex = query.indexOf(',');
         String name = query.substring(
            "find(".length(), commaIndex);
         name = name.trim().toLowerCase();
         String town = query.substring(
            commaIndex+1, query.length()-1);
         town = town.trim().toLowerCase();
         String nameAndTown = combineNameAndTown(town, name);
         printAllMatches(nameAndTown, phoneBook);
      } else {
         System.out.println(query + " is invalid command!");
      }
   }

   private static void printAllMatches(String key,
         HashMap<String, ArrayList<String>> phoneBook) {
      List<String> allMatches = phoneBook.get(key);
      if (allMatches != null) {
         for (String entry : allMatches) {
            System.out.println(entry);
         }
      } else {
```

```
            System.out.println("Not found!");
        }
        System.out.println();
    }
}
```

При прочитането на телефонния указател чрез регулярен израз от него се извличат трите колони (име, град и телефон) от всеки негов ред. След това името се разделя на думи и всяка от думите се добавя в хеш-таблицата. Допълнително се добавя и всяка дума, комбинирана с града (за да можем да търсим по двойката име + град).

Следва втората част на алгоритъма – изпълнението на командите. В нея файлът с командите се чете ред по ред и всяка команда се обработва. Обработката включва парсване на командата, извличането на име или име и град от нея и търсене по даденото име или име, комбинирано с града. Търсенето се извършва директно в хеш-таблицата, която се създава при прочитане на телефонния указател. За да се игнорират разликите между малки и главни букви, всички ключове в хеш-таблицата се добавят с малки букви и при търсенето ключовете се търсят също с малки букви.

Избор на структури от данни – изводи

Видяхте, че изборът на подходяща структура от данни силно зависи от конкретната задача. Понякога се налага структурите от данни да се комбинират или да се използват едновременно няколко структури.

Кога каква структура да подберем зависи най-вече от операциите, които извършваме, така че винаги си задавайте въпроса "какви операции трябва да изпълнява ефективно структурата, която ми трябва". Ако знаете операциите, лесно може да съобразите коя структура ги изпълнява най-ефективно и същевременно е лесна и удобна за ползване.

За да изберете ефективно структура от данни, трябва първо да измислите алгоритъма, който ще имплементирате и след това да потърсите подходящите структури за него.

Тръгвайте винаги от алгоритъма към структурите от данни, а не обратното.

Упражнения

1. Хеш-таблиците не позволяват в един ключ да съхраняваме повече от една стойност. Как може да се заобиколи това ограничение?
2. Реализирайте структура от данни, която изпълнява бързо следните 2 операции: добавяне на елемент; извличане на най-малкия елемент.

Структурата трябва да позволява включването на повтарящи се елементи.

3. Реализирайте структура от данни, която съхранява тройки елементи от вида (key1, key2, value) и позволява да търсим бързо по кой да е измежду двата ключа.
4. В една голяма верига супермаркети се продават милиони стоки. Всяка от тях има уникален номер (баркод), производител, наименование и цена. Каква структура от данни можем да използваме, за да можем бързо да намерим всички стоки, които струват между 5 и 10 лева?
5. Разписанието на дадена конгресна зала представлява списък от събития във формат [начална дата и час; крайна дата и час; наименование на събитието]. Какви структури от данни можем да ползваме, за да можем бързо да проверим дали залата е свободна в даден интервал [начална дата и час; крайна дата и час]?
6. Представете си, че разработвате търсачка в обявите за продажба на коли на старо, която обикаля десетина сайта за обяви и събира от тях всички обяви за последните няколко години. След това търсачката позволява бързо търсене по един или няколко критерии: марка, модел, цвят, година на производство и цена. Нямате право да ползвате система за управление на бази от данни и трябва да реализирате собствено индексиране на обявите в паметта, без да пишете на твърдия диск. При търсене по цена се подава минимална и максимална цена. При търсене по година на производство се задава начална и крайна година. Какви структури от данни ще ползвате, за да осигурите бързо търсене по един или няколко критерия?

Решения и упътвания

1. Можем да използваме **`HashMap<key, List<value>>`**.
2. Можете да използвате **`TreeSet<List<Integer>>`** и неговите операции **`add()`** и **`first()`**. Задачата има и друго решение – структурата от данни "двоична пирамида" (binary heap). Можете да прочетете за нея от Уикипедия: http://en.wikipedia.org/wiki/Binary_heap.
3. Използвайте комбинация от две хеш-таблици.
4. Ако държим стоките в **`TreeMap`**, където за ключ се използва цената, можем да използваме метода **`subMap(5.0, 10.001)`**, за да намерим всички стоки, които струват между 5 и 10 лева. Странното е, че при този метод долната граница се задава включително, а горната – изключително.
5. Можем да конструираме две инстанции на **`TreeMap<date, event>`**, като едната пази събитията по ключ началната дата и час, а другата пази същите събития по ключ крайна дата и час. Можем да намерим всички събития, които се съдържат частично или изцяло между два момента от времето [**`start, end`**] по следния начин:

- Намираме всички събития, завършващи след момента **start** (чрез метода **subMap()**).
- Намираме всички събития, започващи преди момента **end** (чрез метода **subMap()**).
- Ако двете множества от събития имат общи елементи, то в търсения интервал от време [**start, end**] залата е заета. В противен случай залата е свободна.

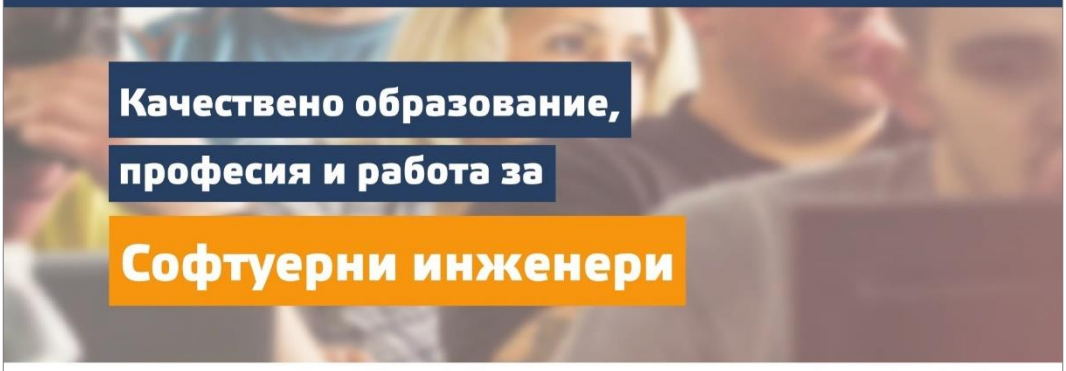
6. За търсенето по марка, модел и цвят можем да използваме по една хеш-таблица, която търси по даден критерий и връща множество от коли (**HashMap<String, HashSet<Car>>**).

   За търсенето по година на производство и по ценови диапазон можем да използваме **TreeMap<String, HashSet<Car>>**. Тази структура ще ни позволи по даден диапазон да намерим съвкупност от множества коли, които влизат в него. Ако обединим множествата, ще получим всички коли в даден диапазон (ценови диапазон или диапазон за годината на производство).

   Ако търсим по няколко критерия едновременно, можем да извлечем множествата коли, по първия критерии, след това множествата коли по втория критерий и т.н. Накрая можем да намерим сечението на множествата. Сечение на две множества се намира, като всеки елемент на по-малкото множество се търси в по-голямото множество.

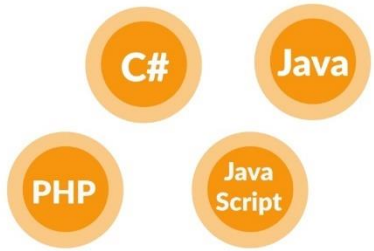
   За класа **Car** ще трябва да дефинираме методите **equals()** и **hashCode()**, за да можем да го ползваме като елемент на множество **HashSet**. Можем да използваме средствата на Eclipse за автоматично генериране на тези методи.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 20. Принципи на обектно-ориентираното програмиране

Автор

Михаил Стойнов

В тази тема...

В настоящата тема ще се запознаем с принципите на обектно-ориентираното програмиране: наследяване на класове и имплементиране на интерфейси, абстракция на данните и поведението, капсулация на данните и скриване на информация за имплементацията на класовете, полиморфизъм и виртуални методи. Ще обясним в детайли принципите за свързаност на отговорностите и функционално обвързване (cohesion и coupling). Ще опишем накратко как се извършва обектно-ориентирано моделиране и как се създава обектен модел по описание на даден бизнес проблем. Ще се запознаем с езика UML и ролята му в процеса на обектно-ориентираното моделиране. Накрая ще разгледаме съвсем накратко концепцията "шаблони за дизайн" и ще дадем няколко типични примера за шаблони, широко използвани в практиката.

Да си припомним: класове и обекти

Класовете са описание (модел) на реални предмети или явления, наречени същности (entities). Например класът "Студент".

Класовете имат характеристики – в програмирането са наречени **свойства (properties)**. Например съвкупност от оценки.

Класовете имат и поведение **–** в програмирането са наречени **методи (methods)**. Например явяване на изпит.

Методите и свойствата могат да бъдат видими и невидими – от това зависи дали всеки може да ги използва или са само за вътрешна употреба в рамките на класа.

Обектите (objects) са екземпляри (инстанции) на класовете. Например Иван е Студент, Петър също е студент.

Обектно-ориентирано програмиране (ООП)

Обектно-ориентираното програмиране е наследник на процецурното (структурно) програмиране. Процедурното програмиране най-общо казано описва програмите чрез група от преизползваеми парчета код (процедури), които дефинират входни и изходни параметри. Процедурните програми представляват съвкупност от процедури, които се извикват една друга.

Проблемът при процедурното програмиране е, че преизползваемостта на кода е трудно постижима и ограничена – само процедурите могат да се преизползват, а те трудно могат да бъдат направени общи и гъвкави. Няма лесен начин да се реализират абстрактни структури от данни, които имат различни имплементации.

Обектно-ориентираният подход залага на парадигмата, че всяка програма работи с данни, описващи същности (предмети и явления) от реалния живот. Например една счетоводна програма работи с фактури, стоки, складове, наличности, продажби и т.н.

Така се появяват обектите – те описват характеристиките (свойства) и поведението (методи) на тези същности от реалния живот.

Основни предимства и цели на ООП – да позволи по-бърза разработка на сложен софтуер и по-лесната му поддръжка. ООП позволява по лесен начин да се преизползва кода, като залага на прости и общоприети правила (принципи). Нека ги разгледаме.

Основни принципи на ООП

За да бъде един програмен език обектно-ориентиран, той трябва не само да позволява работа с класове и обекти, но и трябва да дава възможност за имплементирането и използването на принципите и концепциите на ООП: наследяване, абстракция, капсулация и полиморфизъм. Сега ще разгледаме в детайли всеки от тези основни принципи на ООП.

- **Наследяване (Inheritance)**

  Ще обясним за как йерархиите от класове подобряват четимостта на кода и позволяват преизползване на функционалност.

- **Абстракция (Abstraction)**

  Ще се научим да виждаме един обект само от гледната точка, която ни интересува, и да игнорираме всички останали детайли.

- **Капсулация (Encapsulation)**

  Ще се научим да скриваме ненужните детайли в нашите класове и да предоставяме прост и ясен интерфейс за работа с тях.

- **Полиморфизъм (Polymorphism)**

Ще обясним как да работим по еднакъв начин с различни обекти, които дефинират специфична имплементация на някакво абстрактно поведение.

Наследяване (Inheritance)

Наследяването е основен принцип от обектно-ориентираното програмиране. То позволява на един клас да "наследява" (поведение и характеристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функционалност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н.

Как се дефинира наследяване в Java?

Наследяването в Java става с ключовата дума **`extends`**. В Java и други модерни езици за програмиране един клас може да наследи само един друг клас **(single inheritance)**, за разлика от C++, където се поддържа множествено наследяване **(multiple inheritance)**. Ограничението е породено от това, че при наследяване на два класа с еднакъв метод е трудно да се реши кой от тях да се използва (при C++ този проблем е решен много сложно). В Java могат да се наследяват множество интерфейси, за които ще говорим по-късно.

Класът, който наследяваме, се нарича **клас-родител** или още **базов клас (base class, super class)**.

Наследяване на класове – пример

Да разгледаме един пример за наследяване на класове в Java. Ето как изглежда базовият (родителски) клас:

Felidae.java

```
package introjavabook;

public class Felidae { // Latin word for "cat"

   private boolean male;

   public Felidae() {
      this(true);
   }

   public Felidae(boolean male) {
      this.male = male;
   }

   public boolean isMale() {
```

```
      return male;
   }

   public void setMale(boolean male) {
      this.male = male;
   }
}
```

Ето как изглежда и класът-наследник **Lion**:

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
   private int weight;

   public Lion(boolean male, int weight) {
      super(male); // Shall be explained in the next paragraph
      this.weight = weight;
   }

   public int getWeight() {
      return weight;
   }

   public void setWeight(int weight) {
      this.weight = weight;
   }
}
```

Ключовата дума super

В горния пример в конструктора на **Lion** използваме ключовата дума **super**. Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи. Със **super()** можем да извикваме конструктор на базовия клас. Със **super.method()** можем да извикваме метод на базовия клас, да му подаваме параметри и да използваме резултата от него. Със **super.field** можем да вземем стойността на член-променлива на базовия клас или да й присвоим друга стойност.

В Java наследените от базовия клас методи могат да се **пренаписват** (**override**). Това означава да им се подмени имплементацията, като оригиналният сорс код от базовия клас се игнорира, а на негово място се написва друг код. Повече за пренаписването на методи ще обясним в секцията "Виртуални методи".

Можем да извикваме непренаписан метод от базовия клас и без **super**. Употребата на ключовата дума е необходима само ако имаме пренаписан метод или променлива със същото име в наследения клас.

`super` може да се използва изрично, за яснота. `super.method()` извиква метод, който задължително е от базовия клас. Такъв код се чете по-лесно, защото знаем къде да търсим въпросния метод.

Имайте предвид, че ситуацията с `this` не е такава. `this` може да означава както метод от конкретния клас, така и метод от който и да е базов клас.

Можете да погледнете примера в секцията нива на достъп при наследяване. В него ясно се вижда до кои членове (методи, конструктори и член-променливи) на базовия клас имаме достъп.

Конструкторите при наследяване

При наследяване на един клас, нашите конструктори задължително трябва да извикат конструктор на базовия клас, за да може и той да инициализира член-променливите си. Ако не го направим изрично, в началото на всеки наш конструктор компилаторът поставя извикване на базовия конструктор без параметри: **`super()`**. Ако базовият клас няма конструктор по подразбиране (без параметри), нашите конструктори трябва да извикат изрично някои от другите конструктори на базовия клас. Липсата на изрично извикване предизвиква грешка при компилация.

Конструкторите и super – пример

Разгледайте класа **`Lion`** от последния пример, той няма конструктор по подразбиране. Да разгледаме следния клас-наследник на **`Lion`**:

AfricanLion.java

```
package introjavabook;

public class AfricanLion extends Lion {

   // ...

   public AfricanLion(boolean male, int weight) {
      // If we comment the next line, AfricanLion
      // will not compile. Try it.
      super(male, weight);
   }

   public String toString() {
```

```
      return String.format(
         "(AfricanLion, male: %s, weight: %s)",
         this.isMale(), this.getWeight() );
   }

   // ...
}
```

Ако закоментираме или изтрием реда "**super(male, weight);**", класът **AfricanLion** няма да се компилира. Опитайте.

Извикването на конструктор на базов клас трябва винаги да е на първия ред от нашия конструктор. Иначе компилаторът дава грешка. Идеята е полетата на базовия клас да бъдат инициализирани преди да започнем да инициализираме полета в класа-наследник, защото може те да разчитат на някое поле от базовия клас.

Нива на достъп при наследяване

В главата "Дефиниране на класове" разгледахме нивата на достъп за свойствата и методите: **public**, **private** и **default** (friendly). Освен тях в Java има и още едно ниво на достъп – **protected**. То е свързано с наследяването.

Когато се наследява един базов клас:

- Всички негови **public** и **protected** методи и свойства са видими за класа наследник.
- Всички негови **private** методи и свойства не са видими за класа наследник.
- Всички негови **default** (friendly) методи и свойства са видими за класа наследник само ако базовият клас и наследникът са в един и същ пакет (package).

Ето един пример, с който ще демонстрираме нивата на видимост при наследяване:

Felidae.java

```
package introjavabook;

public class Felidae { // Latin for cat

   private boolean male;

   public Felidae() {
      // Call another constructor with default values
```

```
      this(true);
   }

   public Felidae(boolean male) {
      this.male = male;
   }


   //...
}
```

Ето как изглежда и класът **Lion**:

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
   private int weight;

   public Lion(boolean male, int weight) {
      super(male); // visible - Felidae's public constructor.
      super.male = male; // invsible – male is private.
      this.weight = weight;
   }

   //...
}
```

Ако се опитаме да компилираме този пример, ще получим грешка, тъй като private променливата **male** от класа **Felidae** не е достъпна от класа **Lion**.

Класът Object

Появата на обектно-ориентираното програмиране de facto става популярно с езика C++. В него често се налага да се пишат класове, които трябва да работят с обекти от всякакъв тип. В C++ този проблем се решава по начин, който не се смята за много обектно-ориентиран стил (чрез използване на указатели).

Архитектите на Java поемат в друга посока. Те създават клас, който всички други класове пряко или косвено да наследяват и до който всеки обект може да бъде преобразуван. В този клас е удобно да бъдат сложени важни методи и тяхната имплементация по подразбиране. Този клас се нарича **Object**.

В Java всеки клас, който не наследява друг клас изрично, наследява системния клас **java.lang.Object** по подразбиране. За това се грижи

компилаторът. Всеки клас, който наследява друг клас, наследява индиректно **Object** от него. Така всеки клас явно или неявно наследява **Object** и има в себе си всички негови методи и полета.

Благодарение на това свойство всеки обект може да бъде преобразуван до **Object**. Типичен пример за ползата от неявното наследяване на **Object** е при колекциите, които разгледахме в главите за структури от данни. Списъчните структури (например **ArrayList**) могат да работят с всякакви обекти, защото ги разглеждат като инстанции на класа **Object**.

Java, стандартните библиотеки и Object

В Java има много предварително написани класове (вече разгледахме доста от тях в главите за колекции, текстови файлове и символни низове). Тези класове са част от Java платформата – навсякъде, където има Java, ги има и тях. Тези класове се наричат **стандартни клас-библиотеки – standard class libraries**.

Java е първата платформа, която идва с такъв богат набор от предварително написани класове. Голяма част от тях работят с **Object**, за да могат да бъдат използвани на възможно най-много места.

В Java има и доста библиотеки, които могат да се добавят допълнително и съвсем логично се наричат просто клас-библиотеки или още външни библиотеки.

Object – пример

Нека разгледаме класа **Object** с един пример:

ObjectExample.java

```
package introjavabook;

public class ObjectExample {
   public static void main(String... args) {
      AfricanLion africanLion = new AfricanLion();
      // Implicit casting
      Object object = africanLion;
   }
}
```

В този пример преобразувахме един **AfricanLion** в **Object**. Тази операция се нарича **upcasting** и е позволена, защото **AfricanLion** е непряк наследник на класа **Object**.

Методът Object.toString()

Един от най-използваните методи, идващи от класа **Object,** е **toString()**. Той връща текстово представяне на обекта. Всеки обект има такъв метод и

следователно всеки метод има текстово представяне. Този метод се използва, когато отпечатваме обект чрез **System.out.println()**.

Object.toString() – пример

Ето един пример, в който извикваме **toString()** метода:

ToStringExample.java

```
package introjavabook;

public class ToStringExample {
   public static void main(String... args) {
      AfricanLion africanLion = new AfricanLion();
      System.out.println(africanLion.toString());

      // Result: "introjavabook.AfricanLion@de6ced"
   }
}
```

Тъй като **AfricanLion** не пренаписва (override) метода **toString()**, в конкретния случай се извиква имплементацията от базовия клас. **Lion** и **Felidae** също не пренаписват този метод, следователно се извиква имплементацията, наследена от класа **java.lang.Object**. В резултата, който виждаме по-горе, се съдържа пакетът на обекта, името на класа, както и странна стойност след **@** знака. Това всъщност е хеш кодът на обект в шестнайсетична бройна система. Това не е адресът в паметта, а някаква друга стойност. Обикновено тази стойност е различна за различните оебкти.

Ето я и оригиналната имплементация на метода **Object.toString()**, извадена от сорс кода на стандартните библиотеки в Java:

Object.java

```
Public class Object {
   // ...
   public String toString() {
      return getClass().getName() +
         "@" + Integer.toHexString(hashCode());
   }
   // ...
}
```

Пренаписване на toString() – пример

Нека сега ви покажем колко полезно може да е пренаписването на метода **toString()**, наследено от **java.lang.Object**:

AfricanLion.java

```
public class AfricanLion extends Lion {

   // ...

   public AfricanLion(boolean male, int weight) {
      super(male, weight);
   }

   public String toString() {
      return String.format(
         "(AfricanLion, male: %s, weight: %s)",
         this.isMale(), this.getWeight());
   }

   // ...
}
```

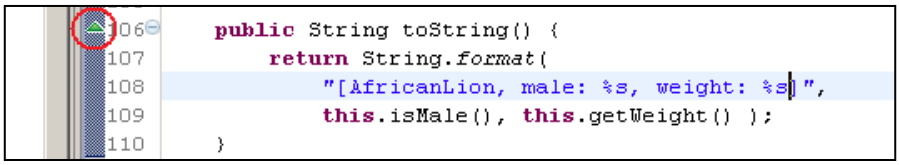
В горния код използваме **String.format(String format, Object... args)** метода, за да форматираме резултата по подходящ начин. Ето как можем след това да извикваме пренаписания метод **toString()**:

ToStringExample.java

```
package introjavabook;

public class ToStringExample {

   public static void main(String... args) {

      AfricanLion africanLion = new AfricanLion(true, 15);
      System.out.println(africanLion);

      // Result: "[AfricanLion, male: true, weight: 15]"
   }
}
```

Забележете, че извикването на **toString()** става скрито. Когато на метода **println()** подадем някакъв обект, този обект първо се преобразува до стринг чрез **toString()** метода му и след това се отпечатва в изходния поток. Така при печатане на конзолата няма нужда изрично да преобразуваме обектите до стринг.

Ако ползваме средата Eclipse и искаме да сме сигурни, че пренаписваме метод, можем да следим за зелен триъгълник, който показва, че нашият метод пренаписва друг:



От Java 5 нататък има начин да укажем изрично на компилатора, че искаме нашият метод да пренаписва друг. За целта се използват се т. нар. **анотации**, а в конкретния случай се използва анотацията **@Override**:

AfricanLion.java

```
public class AfricanLion extends Lion {
   // ...

   @Override
   public String toString() {
      return String.format(
         "(AfricanLion, male: %s, weight: %s)",
         this.isMale(), this.getWeight());
   }

   // ...
}
```

Изричното указване на компилатора, че искаме да пренапишем метод от базов клас, е препоръчителна практика и намалява грешките. Ако си създадем навика при пренаписване на метод винаги да ползваме анотацията **@Override**, ако случайно сбъркаме една буква от името на метода или типовете на неговите параметри, компилаторът веднага ще ни съобщи за грешката.

Транзитивност при наследяването

В математиката транзитивност означава прехвърляне на взаимоотношения. Нека вземем операцията "по-голямо". Ако A>B и B>C, то можем да заключим, че A>C. Това означава, че релацията "по-голямо" (>) е транзитивна, защото може еднозначно да бъде определено дали A е по-голямо от C или обратното.

Ако клас **Lion** наследява клас **Felidae**, а клас **AfricanLion** наследява клас **Lion**, това индиректно означава, че **AfricanLion** наследява **Felidae**. Следователно **AfricanLion** също има свойство **male**, което е дефинирано във

Felidae. Това полезно свойство позволява определена функционалност да бъде описана в най-подходящия за нея клас.

Транзитивност – пример

Ето един пример, който демонстрира транзитивността при наследяване:

TransitiveInheritance.java

```
package introjavabook;

public class TransitiveInheritance {

   public static void main(String... args) {

      AfricanLion africanLion = new AfricanLion(true, 15);
      // Method defined in Felidae
      africanLion.isMale();
      // Method defined in Felidae
      africanLion.setMale(true);
   }
}
```

Заради транзитивността на наследяването можем да сме сигурни, че всички класове имат **toString()** и другите методи на **Object** без значение кой клас наследяват.

Йерархия на наследяване

Ако тръгнем да описваме всички големи котки, рано или късно се стига до сравнително голяма група класове, които се наследяват един друг. Всички тези класове, заедно с базовите такива, образуват йерархия от класове на големите котки. Такива йерархии могат да се опишат най-лесно чрез клас-диаграми. Нека разгледаме какво е това "клас-диаграма".

Клас-диаграми

Клас-диаграмата е един от няколко вида диаграми дефинирани в UML. **UML (Unified Modeling Language)** е нотация за визуализация на различни процеси и обекти, свързани с разработката на софтуер. Обяснена е по-подробно към края на тази глава. Сега, нека ви разкажем малко за клас-диаграмите, защото те се използват, за да описват визуално йерархиите от класове, наследяването и вътрешността на самите класове.

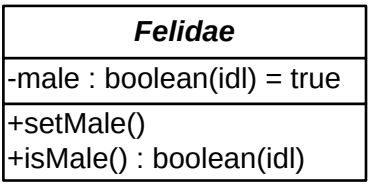
В клас-диаграмите има възприети правила класовете да се рисуват като правоъгълници с име, атрибути (член-променливи) и операции (методи), а връзките между тях се обозначават с различни видове стрелки.

Накратко ще обясним два термина от UML, за по ясно разбиране на примерите. Единият е **генерализация (generalization)**. Генерализация е

обобщаващо понятие за наследяване на клас или имплементация на интерфейс (за интерфейси ще обясним след малко). Другият термин се нарича **асоциация (association)**. Например "Лъвът има лапи", където **Лапа** е друг клас. Генерализация и асоциация са двата най-основни начина за преизползване на код.

Един клас от клас диаграма – пример

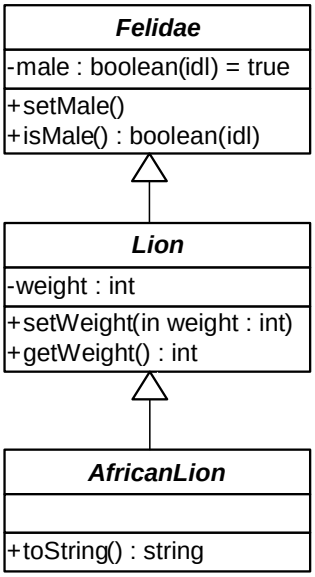
Ето как изглежда една примерна клас-диаграма на един клас:



Класът е представен като правоъгълник, разделен на 3 части, разположени една под друга. В най-горната част е дефинирано името на класа. В следващата част след него са атрибутите (термин от UML) на класа (в Java се наричат член-променливи и свойства). Най-отдолу са операциите (в UML) или методите (в Java). Плюсът/минусът в началото указват дали атрибутът/операцията са видими (`+` означава `public`) или невидими (`-` означава `private`). `Protected` членовете се означават със символа `#`.

Клас-диаграма – генерализация – пример

Ето пример за клас диаграма, показваща генерализация:



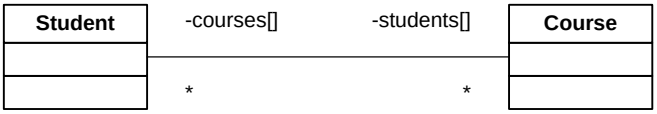
В този пример стрелките означават генерализация или наследяване.

Асоциации

Асоциациите представляват връзки между класовете. Те моделират взаимоотношения. Могат да дефинират множественост (1 към 1, 1 към много, много към 1, 1 към 2, ..., и много към много).

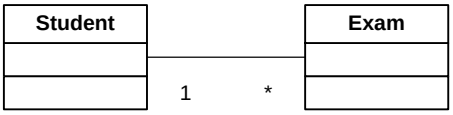
Асоциация **много към много (many-to-many)** се означава по следния начин:

Асоциация **много към много (many-to-many) по атрибут** се означава по следния начин:

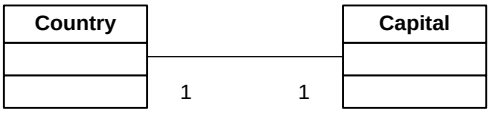


В този случай има свързващи атрибути, които показват в кои променливи се държи връзката между класовете.

Асоциация **едно към много (one-to-many)** се означава така:



Асоциация **едно към едно (one-to-one)** се означава така:



От диаграма към класове

От клас-диаграмите най-често се създават класове. Диаграмите улесняват и ускоряват дизайна на класовете на един софтуерен проект.

От горната диаграма можем директно да създадем класове. Ето класът **`Country`**:

Country.java

```
package introjavabook;

public class Country {

   /** Country's capital. */
   private Capital capital;
```

```
   // ...

   public Capital getCapital() {
      return capital;
   }

   public void setCapital(Capital capital) {
      this.capital = capital;
   }

   // ...
}
```

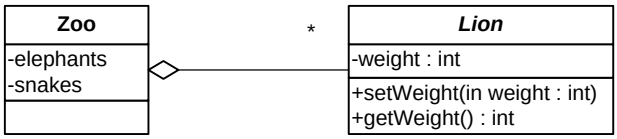
Ето и класа **Capital**:

Capital.java

```
package introjavabook;


public class Capital { }
```

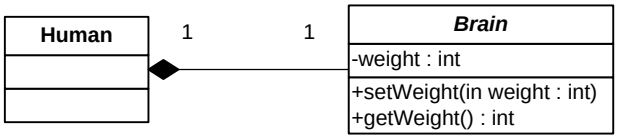
Агрегация

Агрегацията е специален вид асоциация. Тя моделира връзката "цяло / част". **Агрегат** наричаме родителския клас. **Компоненти** наричаме агрегираните класове. В единия край на агрегацията има празен ромб:



Композиция

Запълнен ромб означава композиция. Композицията е агрегация, при която компонентите не могат да съществуват без агрегата (родителя):



Абстракция (Abstraction)

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". **Абстракцията** означава да работим

с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното, и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.

Същото се получава и с обектите в ООП. Ако имаме обект **Лаптоп** и той се нуждае от процесор, просто използваме обекта **Процесор**. Не знаем (или поточно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода **сметни()** с подходящи параметри.

Абстракцията е нещо, което правим всеки ден. Това е действие, при което игнорираме всички детайли, които не ни интересуват от даден обект и разглеждаме само детайлите, които имат значение за проблема, който решаваме. Например в хардуера съществува абстракция "устройство за съхранение на данни", което може да бъде твърд диск, USB memory stick, флопи диск или CD-ROM устройство. Всяко от тях работи вътрешно по различен начин, но от гледна точка на операционната система и на програмите в нея те се използват по еднакъв начин – на тях се записват файлове и директории. В Windows имаме Windows Explorer и той умее да работи по еднакъв начин с всички устройства, независимо дали са твърд диск или USB stick. Той работи с абстракцията "устройство за съхранение на данни" (storage device) и не се интересува как точно данните се четат и пишат. За това се грижат драйверите за съответните устройства. Те се явяват конкретни имплементации на интерфейса "устройство за съхранение на данни".

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем **код, който работи с абстрактни структури от данни** (например списък, речник, множество и други). Имайки абстрактния тип данни ние можем да работим с него през неговия интерфейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързана имплементация или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – **да дефинираме интерфейс на нашите програми**, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Това, допълнено от факта, че можем да работим с абстрактни данни, ни дава голяма гъвкавост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпрограми се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми.

Абстракция – пример за абстрактни данни

Ето един пример, в който дефинираме конкретен тип данни "африкански лъв", но след това го използваме по абстрактен начин – чрез абстракцията "лъв". Тази абстракция не се интересува от детайлите на всички видове лъвове.

AbstractDataExample.java

```
package introjavabook;

public class AbstractDataExample {
   public static void main(String... args) {

      Lion lion = new Lion(true, 150);
      Felidae bigCat1 = lion;

      AfricanLion africanLion = new AfricanLion();
      Felidae bigCat2 = africanLion;
   }
}
```

Интерфейси

В езика Java **интерфейсът** е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект **Човек** може да има ролите **Военен** (с поведение "стреляй по противника"), **Съпруг** (с поведение "обичай жена си"), **Данъкоплатец** (с поведение "плати си данъка"). Всеки човек обаче имплементира това поведение по различен начин: **Иван** си плаща данъците навреме, **Георги** – не навреме, **Петър** – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас **Object** не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация, тоест имплементацията по подразбиране върши работа в повечето случаи. От класа **Object** не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли след малко.

Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

Декларация на метод (method declaration) е съвкупността от връщания тип на метода + сигнатурата на метода. Връщаният тип е просто за яснота какво ще върне метода.

Сигнатура на метод (method signature) е съвкупността от името на метода + описание на параметрите (тип и последователност). В един клас/интерфейс всички методи трябва да са с различни сигнатури и да не съвпадат със сигнатури на наследени методи.

Това, което идентифицира един метод, е неговата сигнатура. Връщаният тип не е част нея. Причината е, че ако два метода се различават само по връщания тип (например два класа, които се наследяват един друг), то не може еднозначно да се идентифицира кой метод трябва да се извика.

Имплементация на клас/метод (class/method implementation) е тялото със сорс код на класа/метода. Най често е заключено между скобите **{** и **}**. При методите се нарича още **тяло на метод**.

Интерфейси – пример

Интерфейсът в Java се дефинира с ключовата думичка **`interface`**. В него може да има само декларации на методи, както и статични променливи (за константи например). Ето един пример за интерфейс:

Reproducible.java

```
package introjavabook;

public interface Reproducible {
   Mammal[] reproduce(Mammal mate);
}
```

Ето как изглежда и класа **`Lion`**, който имплементира интерфейса **`Reproducible`**:

Lion.java

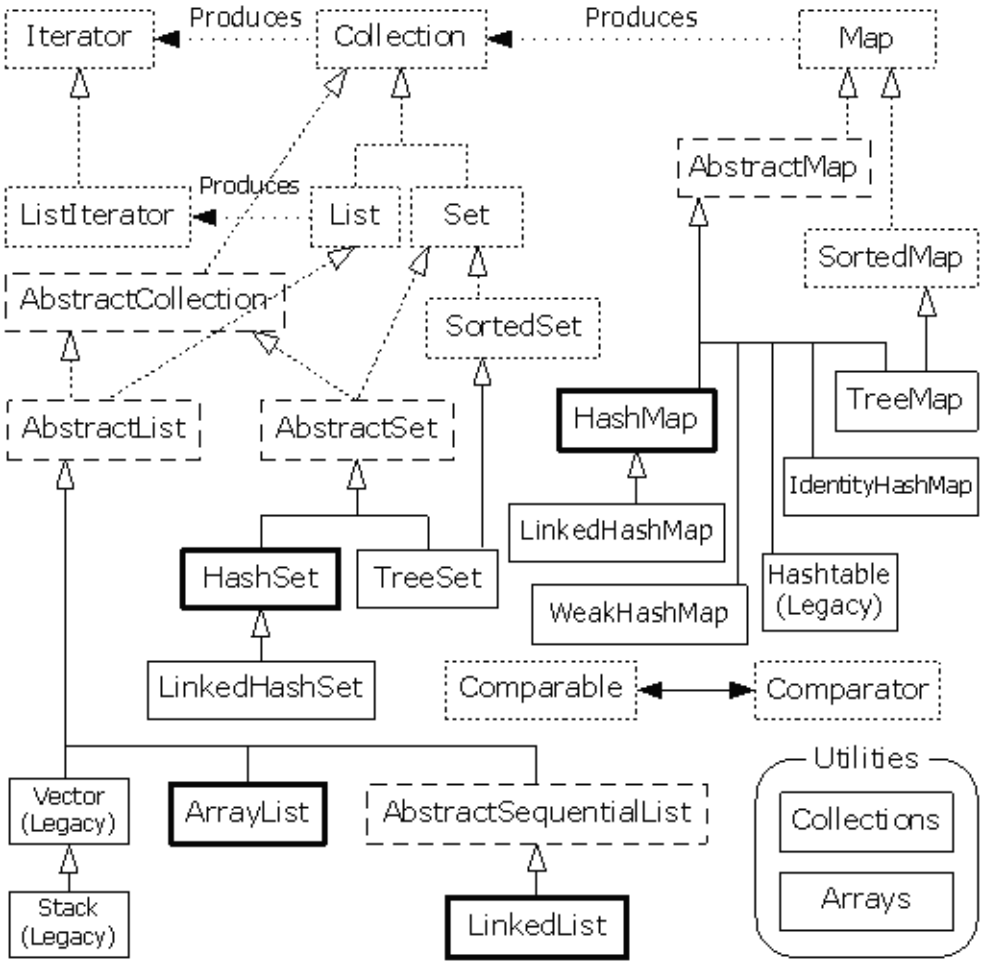
```
package introjavabook;

public class Lion extends Felidae implements Reproducible {
   // ...

   public Mammal[] reproduce(Mammal anotherLion) {
      return new Mammal[]{new Lion(), new Lion()};
   }
}
```

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса - **`Lion`**.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него. Изключение – ако класът е абстрактен, тогава да имплементира нула, няколко или всички методи. Всички останали методи имплементират в някой от класовете наследници.

Абстракция и интерфейси

Най-добрият начин да се реализира абстракция е да се работи с интерфейси. Един компонент работи с интерфейси, които друг имплементира. Така подмяната на втория компонент няма да се отрази на първия, стига новият компонент да имплементира старите интерфейси. Интерфейсът се нарича още **договор** (**contract**). Всеки компонент, имплементирайки един интерфейс, спазва определен договор. Така два компонента, стига да спазват правилата на договора, могат да общуват един с друг, без да знаят как работи другата страна.



Примери за важни интерфейси от Java collections framework са **java.util.List** и **java.util.Collection**. Всички стандартни колекции имплементират тези интерфейси и различните компоненти си прехвърлят различни имплементации (масиви или свързани списъци, хеш-таблици, червено-черни дървета и др.) винаги под общ интерфейс. На фигурата по-горе е показано как изглежда част от йерархията на колекциите в Java.

Колекциите са един отличен пример на обектно-ориентирана библиотека с класове и интерфейси, при която се използват много активно всички основни принципи на ООП: абстракция, наследяване, капсулация и полиморфизъм. От картинката се вижда, че абстрактните типове данни са дефинирани като интерфейси (**Collection**, **List**, **Set**, **Map** и други), а конкретните им имплементации са техни преки или непреки наследници в йерархията (**ArrayList**, **LinkedList**, **HashSet**, **HashMap**, **TreeMap** и други).

Кога да използваме абстракция и интерфейси?

Отговорът на този въпрос е: винаги, когато искаме да постигнем абстракция на данни или действия, чиято имплементация по-късно може да се подмени. Написаният код срещу интерфейси е много по-издръжлив срещу промени, отколкото написаният срещу конкретни класове. Работата през интерфейси е често срещана и силно препоръчвана практика – едно от основните правила за писане на качествен код.

Кога да пишем интерфейси?

Винаги е добра идея да се използват интерфейси, когато се предоставя функционалност на друг компонент. В интерфейса се слага само функционалността (като декларация), която другите трябва да виждат.

Вътрешно в една програма/компонент интерфейсите могат да се използват за дефиниране на роли. Така един обект може да се използва от много класове чрез различните му роли.

Капсулация (Encapsulation)

Капсулацията е един от основните принципи на обектно-ориентираното програмиране. Тя се нарича още "скриване на информацията" (**information hiding**). Един обект трябва да предоставя на ползвателя си само необходимите средства за управление. Една **Секретарка** ползваща един **Лаптоп** знае само за екран, клавиатура и мишка, а всичко останало е скрито. Тя няма нужда да знае за вътрешността на **Лаптопа**, защото не ѝ е нужно и може да оплеска нещо. Тогава част от свойствата и методите остават скрити за нея.

Изборът какво е скрито и какво е публично видимо е на този, който пише класа. Когато програмираме трябва да дефинираме като private (скрит) всеки метод или поле, които не ползваме от друг клас.

Капсулация – примери

Ето един пример за скриване на методи, които не е нужда да са известни на потребителя, а се ползват вътрешно само от автора на класа. Първо дефинираме абстрактен клас **Felidae**, който дефинира публичните операции на котките (независимо какви точно котки имаме):

Felidae.java

```
package introjavabook;

public abstract class Felidae { // Latin for cat
   // ...

   public abstract void walk();
}
```

Ето как изглежда класът **Lion**:

Lion.java

```
package introjavabook;

public class Lion extends Felidae implements Reproducable {
   // ...

   private movePaw(Paw paw) {
      // ...
   }

   @Override
   public void walk() {
      this.movePaw(frontLeft);
      this.movePaw(frontRight);
      this.movePaw(bottomLeft);
      this.movePaw(bottomRight);
   }
}
```

Публичният метод **walk()** извиква 4 пъти някакъв друг скрит (**private**) метод. Така интерфейсът (в този случай абстрактният клас) е кратък – само един метод. Имплементацията обаче извиква друг метод, също част от имплементацията, но скрит за ползвателя на класа. Така класът **Lion** не разкрива публично информация за това как работи вътрешно и това му дава възможност на по-късен етап да промени имплементацията си без останалите класове да разберат (и да имат нужда от промяна).

Друг пример за абстракция е класът **ArrayList** от стандартните библиотеки на Java. Ако отворим сорс кода на този клас, ще видим, че в него има

десетки полета и методи, които са дефинирани като **private** (скрити) и са достъпни само вътрешно от класа:

ArrayList.java

```
package java.util;

public class ArrayList<E> extends AbstractList<E>
   implements List<E>, RandomAccess, Cloneable, Serializable {
   private static final long serialVersionUID =
      8683452581122892189L;
   private transient Object[] elementData;
   private int size;

   private void fastRemove(int index) { ... }
   private void rangeCheck(int index) { ... }
   private void writeObject(ObjectOutputStream s) { ... }
   private void readObject(ObjectInputStream s) { ... }
}
```

Както виждаме, освен познатите ни публични методи в класа **ArrayList** има и скрити неща. Това са вътрешните структури, съхраняващи елементите на структурата (**elementData** и **size**) и някои тайни методи, които не би трябвало да се извикват извън класа. Скриването на тези детайли гарантира, че никой освен самия клас **ArrayList** няма да бута директно по данните и така няма да има възможност да сбърка нещо. Ако всички полета в **ArrayList** бяха дефинирани като публични, щеше да е много трудно да накараме потребителите да обновяват в синхрон променливите **size** и **elementData**. Понеже тези променливи са скрити, класът **ArrayList** се грижи вътрешно за тях и няма опасност някой да ги разбута.

Полиморфизъм (Polymorphism)

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм". **Полиморфизмът** позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с пренаписването (override) на методи в наследените класове с цел промяна на оригиналното им поведение, наследено от базовия клас. Абстракцията се свързва със създаването на интерфейс на

компонент или функционалност (дефиниране на роля). Пренаписването на методи ще разгледаме в детайли след малко.

Абстрактни класове

Какво става, ако искаме да кажем, че класът **Felidae** е непълен и само наследниците му могат да имат инстанции? Това става с ключовата дума **abstract** пред името на класа и означава, че класът не е готов и не може да бъде инстанциран. Такъв клас се нарича **абстрактен клас**. А как да укажем коя точно част от класа не е пълна? Това отново става с ключовата дума **abstract** пред името на метода, който трябва да бъде имплементиран. Този метод се нарича **абстрактен метод** и не може да притежава имплементация, а само декларация.

Всеки клас, който има поне един абстрактен метод, трябва да бъде абстрактен. Логично, нали? Обратното, обаче не е в сила. Възможно е да дефинирам клас като абстрактен дори когато в него няма нито един абстрактен метод.

Абстрактните класове са нещо средно между клас и интерфейс. Те могат да дефинират обикновени методи и абстрактни методи. Обикновените методи имат тяло (имплементация), докато абстрактните методи са празни (без имплементация) и са оставени да бъдат реализирани от класовете-наследници.

Абстрактен клас – примери

Да разгледаме един пример за абстрактен клас:

Felidae.java

```
package introjavabook;

public abstract class Felidae {// Latin for cat

   // ...

   public boolean isMale() {
      return male;
   }

   public void setMale(boolean male) {
      this.male = male;
   }

   public abstract boolean catchPray(Object pray);
}
```

Забележете в горния пример как нормалните методи **isMale()** и **setMale()** имат тяло, а абстрактният метод **catchPray()** няма тяло.

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
   // ...

   public boolean catchPray(Object pray) {
      super.hide();
      this.ambush();
      super.run();
      // ...
   }
}
```

Ето още един пример за абстрактно поведение, реализирано чрез абстрактен клас и полиморфно извикване на абстрактен метод. Първо дефинираме абстрактния клас **Animal**:

Animal.java

```
package introjavabook;

public abstract class Animal {

   public void printInformation() {
      System.out.println("I am " +
         this.getClass().getSimpleName() + ".");
      System.out.println(getTypicalSound());
   }

   protected abstract String getTypicalSound();
}
```

Дефинираме и класа **Cat**, който наследява абстрактния клас **Animal** и дефинира имплементация за абстрактния метод **getTypicalSound()**:

Cat.java

```
package introjavabook;

public class Cat extends Animal {

   @Override
   protected String getTypicalSound() {
      return "Miaoooow!";
   }
```

```
    public static void main(String[] args) {
        Animal cat = new Cat();
        cat.printInformation();
        // Output:
        //   I am Cat.
        //   Miaoooow!
    }
}
```

В примера методът **`printInformation()`** от абстрактния клас свършва своята работа като разчита на резултата от извикването на абстрактния метод **`getTypicalSound()`**, който се очаква да бъде имплементиран по различен начин за различните животни (различните наследници на класа **`Animal`**). Различните животни издават различни звуци, но отпечатването на информация за животно е една и съща функционалност за всички животни и затова е изнесена в базовия клас.

Чист абстрактен клас

Абстрактните класове, както и интерфейсите не могат да се инстанцират. Ако се опитате да създадете инстанция на абстрактен клас, ще получите грешка по време на компилация. Понякога даден клас може да бъде деклариран като абс**трактен дори и да няма нито един абстрактен метод**, просто, за да се забрани директното му използване, без да се създава инст**анция на негов наследник.**

Чист абстрактен клас (pure abstract class) е абстрактен клас, който няма нито един имплементиран метод, както и нито една член променлива. Много напомня на интерфейс. Основната разлика е, че един клас може да имплементира много интерфейси и наследява само един клас (бил той и чист абстрактен клас).

В началото при съществуването на множествено наследяване не е имало нужда от интерфейси. За да бъде заместено, се е наложило да се появят интерфейсите, които да носят многото роли на един обект.

Виртуални методи

Метод, който може да се пренапише в клас наследник, се нарича **виртуален метод (virtual method)**. Всички методи в Java са виртуални, без изрично да се дефинират като такива. Ако не искаме да бъдат виртуални, ги маркираме с ключовата дума **`final`**. Тогава никой клас-наследник не може да декларира и дефинира метод със същата сигнатура.

Виртуалните методи са важни за **пренаписването на методи (method overriding)**, което е в сърцето на полиморфизма.

Виртуални методи – пример

Имаме клас, наследяващ друг, като и двата имат общ метод. И двата метода пишат на конзолата. Ето как изглежда класът **Lion**:

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
   // ...

   public void catchPray(Object pray) {
      System.out.println("Lion.catchPray");
   }
}
```

Ето как изглежда и класът **AfricanLion**:

AfricanLion.java

```
package introjavabook;

public class AfricanLion extends Lion {
   // ...

   public void catchPray(Object pray) {
      System.out.println("AfricanLion.catchPray");
   }
}
```

Правим три опита за създаване на инстанции и извикване на метода **catchPray**.

VirtualMethodsExample.java

```
package introjavabook;

public class VirtualMethodsExample {

   public static void main(String... args) {
      {
         Lion lion = new Lion();
         lion.catchPray(null);
         // Will print "Lion.catchPray"
      }

      {
```

```
        AfricanLion lion = new AfricanLion();
        lion.catchPray(null);
        // Will print "AfricanLion.catchPray"
     }

     {
        Lion lion = new AfricanLion();
        lion.catchPray(null);
        // Will print "AfricanLion.catchPray", because
        // the variable lion has value of type AfricanLion
     }
   }
}
```

В последния опит ясно се вижда как всъщност се извиква пренаписаният метод, а не базовият. Това се случва, защото се проверява кой всъщност е истинският клас, стоящ зад променливата, и се проверява дали той има имплементиран (пренаписан) този метод.

Пренаписването на методи се нарича още: **припокриване (подмяна) на виртуален метод**.

Както виртуалните, така и абстрактните методи могат да бъдат припокривани. Абстрактните методи всъщност представляват виртуални методи без конкретна имплементация. Всички методи, които са дефинирани в даден интерфейс са абстрактни и следователно виртуални, макар и това да не е дефинирано изрично.

Виртуални методи и скриване на методи

В горния пример имплементацията на базовия клас остана скрита и неизползвана. Ето как можем да ползваме и нея като част от новата имплементация (в случай че не искаме да подменим, а само да допълним старата имплементация):

Lion.java

```
package introjavabook;

public class Lion extends Felidae {
   // ...

   public void catchPray(Object pray) {
      System.out.println("Lion.catchPray");
   }
}
```

Ето как изглежда и класът **AfricanLion**:

AfricanLion.java

```
package introjavabook;

public class AfricanLion extends Lion {
   // ...

   public boolean catchPray(Object pray) {
      System.out.println("AfricanLion.catchPray");
      System.out.println("calling super.catchPray(pray)");
      super.catchPray(pray);
   }
}
```

В този пример при извикването на **AfricanLion.catchPray(…)** ще се изпишат 3 реда на конзолата:

```
AfricanLion.catchPray
calling super.catchPray(pray)
Lion.catchPray
```

Кога да използваме полиморфизъм?

Отговорът на този въпрос е прост: винаги, когато искаме да предоставим възможност имплементацията на даден метод да бъде подменен в клас-наследник. Добро правило е да се работи с възможно най-базовия клас или направо с интерфейс. Така промените върху използваните класове се отразяват в много по-малка степен върху класовете, които ние пишем. Колкото по-малко знае една програма за обкръжаващите я класове, толкова по-малко промени (ако въобще има някакви) трябва да претърпи тя.

Свързаност на отговорностите и функционално обвързване

Термините cohesion и coupling са неразривно свързани с ООП. Те допълват и дообясняват някои от принципите, които описахме до момента. Нека се запознаем с тях.

Свързаност на отговорностите (cohesion)

Понятието **cohesion (свързаност на отговорностите)** показва до каква степен различните задачи и отговорности на една програма или един компонент са свързани помежду си, т.е. колко фокусирана е програмата в решаването на една единствена задача. Разделя се на **силна свързаност (strong cohesion)** и **слаба свързаност (weak cohesion)**.

Силна свързаност на отговорностите (strong cohesion)

Когато кохезията (cohesion) е силна, това показва, че отговорностите и задачите на една единица код (метод, клас, компонент, подпрограма) са свързани помежду си и се стремят да решат общ проблем. Това е нещо, към което винаги трябва да се стремим. Strong cohesion е типична характеристика на висококачествения софтуер.

Силна свързаност за клас

Силна свързаност на отговорностите (strong cohesion) в един клас означава, че този клас описва само един субект. По-горе споменахме, че един субект може да има много роли (Петър е военен, съпруг, данъкоплатец). Всички тези роли се описват в един и същ клас. Силната свързаност означава, че класът решава една задача, един проблем, а не много едновременно. Клас, който прави много неща едновременно е труден за разбиране и поддръжка. Представете си клас, който реализира едновременно хеш-таблица, предоставя функции за печатане на принтер, за пращане на e-mail и за работа с тригонометрични функции. Какво име ще дадем на този клас? Ако се затрудняваме в отговора на този въпрос, това означава, че нямаме силна свързаност на отговорностите (cohesion) и трябва да разделим класа на няколко по-малки, всеки от които решава само една задача.

Силна свързаност за клас – пример

Като пример за силна свързаност на отговорности можем да дадем класа **`java.lang.Math`**. Той изпълнява една единствена задача – предоставя математически изчисления и константи:

- **`Sin()`**, **`Cos()`**, **`Asin()`**
- **`Sqrt()`**, **`Pow()`**, **`Exp()`**
- **`Math.PI`**, **`Math.E`**

Силна свързаност за метод

Един метод е добре написан, когато изпълнява само една задача и я изпълнява добре. Метод, който прави много неща, свързани със съвсем различни задачи, има лоша кохезия и трябва да се раздели на няколко по-прости метода, които решават само една задача. И тук стои въпросът какво име ще дадем на метод, който търси прости числа, чертае 3D графика на екрана, комуникира по мрежата и печата на принтер справки, извлечени от база данни. Такъв метод има лоша кохезия и трябва да се раздели логически на няколко метода.

Слаба свързаност на отговорностите (weak cohesion)

Слаба свързаност се наблюдава при методи, които вършат по няколко задачи. Тези методи трябва да приемат няколко различни групи параметри, за да извършат различните задачи. Понякога това налага несвързани

логически данни да се обединяват за точно такива методи. Използването на слаба кохезия (weak cohesion) е вредно и трябва да се избягва!

Слаба свързаност на отговорностите – пример

Ето един пример за клас, който има weak cohesion:

```
public class Magic {
   public void PrintDocument(Document d) { ... }
   public void SendEmail(string recipient,
      string subject, string text) { ... }
   public void CalculateDistanceBetweenPoints(
      int x1, int y1, int x2, int y2) { ... }
}
```

Добри практики за свързаност на отговорностите

Съвсем логично силната свързаност е "добрият" начин на писане на код. Понятието се свързва с по-прост и по-ясен сорс код – код, който по-лесно се поддържа и по-лесно се преизползва (поради по-малкия на брой задачи, които той изпълнява).

Обратно, при слаба свързаност всяка промяна е бомба със закъснител, защото може да засегне друга функционалност. Понякога една логическа задача се разпростира върху няколко модула и така промяната й е по-трудоемка. Преизползването на код също е трудно, защото един компонент върши няколко несвързани задачи и за да се използва отново, трябва да са на лице точно същите условия, което трудно може да се постигне.

Функционално обвързване (coupling)

Функционално обвързване (coupling) описва най-вече до каква степен компонентите / класовете зависят един от друг. Дели се на **функционална независимост (loose coupling)** и **силна взаимосвързаност (tight coupling)**. Функционалната независимост обикновено идва заедно със слабата свързаност на отговорностите и обратно.

Функционална независимост (loose coupling)

Функционалната независимост (loose coupling) се характеризира с това, че единиците код (подпрограма / клас / компонент) общуват с други такива през ясно дефинирани интерфейси (договори) и промяната в имплементацията на един компонент не се отразява на другите, с които той общува. Когато пишете програмен код, не трябва да разчитате на вътрешни характеристики на компонентите (специфично поведение, неописано в интерфейсите).

Договорът трябва да е максимално опростен и да дефинира единствено нужните за работата на този компонент поведения, като скрива всички ненужни детайли.

Функционалната независимост е характеристика на кода, към която трябва да се стремите. Тя е една от отличителните черти на качествения програмен код.

Loose coupling – пример

Ето един пример, в който имаме функционална независимост между класовете и методите:

```
class Report {
   public boolean loadFromFile(String fileName) {…}

   public boolean saveToFile(String fileName) {…}
}

class Printer {
   public static int print(Report report) {…}
}

class Example {
   public static void main(String[] args) {
      Report myReport = new Report();
      myReport.loadFromFile("DailyReport.xml");
      Printer.print(myReport);
   }
}
```

В този пример никой клас и никой метод не зависи от останалите. Методите зависят само от параметрите, които им се подават. Ако някой метод ни потрябва в следващ проект, лесно ще можем да го извадим и използваме отново.

Силна взаимосвързаност (tight coupling)

Силна взаимосвързаност имаме при много входни параметри и изходни параметри и при използване на неописани (в договора) характеристики на друг компонент (например зависимост от статични полета в друг клас). При използване на много т. нар. контролни променливи, които оказват какво да е поведението със същинските данни. Силната взаимосвързаност между два или повече метода, класа или компонента означава, че те не могат да работят независимо един от друг и че промяната в един от тях ще засегне и останалите. Това води до труден за четене код и големи проблеми при поддръжката му.

Tight coupling – пример

Ето един пример, в който имаме силна взаимосвързаност между класовете и методите:

```
class MathParams {
   public static double operand;
   public static double result;
}

class MathUtil {
   public static void sqrt() {
      MathParams.result = calcSqrt(MathParams.operand);
   }
}

class SpaceShuttle {
   public static void main(String[] args) {
      MathParams.operand = 64;
      MathUtil.sqrt();
      System.out.println(MathParams.result);
   }
}
```

Такъв код е труден за разбиране и за поддръжка, а възможността за грешки при използването му е огромна. Помислете какво се случва, ако друг метод, който извиква **sqrt()** подава параметрите си през същите статични променливи **operand** и **result**.

Ако се наложи в следващ проект да използваме същата функционалност за извличане на корен квадратен, няма да можем просто да си копираме метода **sqrt()**, а ще трябва да копираме класовете **MathParams** и **MathUtil** заедно с всичките им методи. Това прави кода труден за преизползване.

Всъщност горният код е пример за лош код по всички правила на процедурното и обектно-ориентираното програмиране и ако се замислите, сигурно ще се сетите за още поне няколко неспазени препоръки, които сме ви давали до момента.

Добри практики за функционално обвързване

Добрата практика е да не се разчита на нищо повече от описаното в договора (интерфейса). Разбира се, добра практика е да се програмира срещу интерфейси, а не срещу конкретни класове (за това вече споменахме в секцията "Абстракция").

Добра практика е методите да са гъвкави и да са готови да работят с всички компоненти, които спазват интерфейса им, а не само с определени такива. Последното би означавало, че тези методи очакват нещо специфично от компонентите, с които могат да работят. Добра практика е също всички зависимости да са ясно описани и видими. Иначе поддръжката на такъв код става трудна (пълно е с подводни камъни).

Добър пример за strong cohesion и loose coupling е библиотеката Java collections framework (колекциите в Java). Класовете за работа с колекции

имат силна кохезия. Всеки от тях решава една задача и позволява лесна преизползваемост. Тези класове притежават и другата характеристика на качествения програмен код: loose coupling. Класовете, реализиращи колекциите са необвързани един с друг. Всеки от тях работи през строго дефиниран интерфейс и не издава детайли за своята имплементация. Всички методи и полета, които не са от интерфейса, са скрити, за да се намали възможността за обвързване на други класове с тях. Методите в класовете за колекции не зависят от статични променливи и не разчитат на никакви входни данни, освен вътрешното си състояние и подадените им параметри. Това е добрата практика, до която рано или късно всеки програмист достига като понатрупа опит.

Код като спагети (spaghetti code)

Спагети код е неструктуриран код с неясна логика, труден за четене, разбиране и за поддържане. Това е код, в който последователността е нарушена и объркана. Това е код, който има weak cohesion и tight coupling. Този код се свързва се със спагети, защото също като тях е оплетен и завъртян. Като дръпнеш един спагет (т. е. един клас или метод), цялата чиния спагети може да се окаже, оплетена в него (т. е. промяна на един метод или клас води до още десетки други промени поради силната зависимост между тях). Спагети кодът е почти невъзможно да се преизползва, защото няма как да отделиш тази част от него, която върши работа.

Спагети кодът се получава, когато сте писали някакъв код, след това сте го допълнили, след това изискванията са се променили и вие сте нагодили кода към тях, след това пак са се пременили и т.н. С времето спагетите се оплитат все повече и повече и идва момент, в който всичко трябва да се пренапише от нулата.

Cohesion и coupling в инженерните дисциплини

Ако си мислите, че принципите за strong cohesion и loose coupling се отнасят само за програмирането, дълбоко се заблуждавате. Това са здрави

инженерни принципи, които ще срещнете в строителството, в машиностроенето, в електрониката и на още хиляди места.

Да вземем за пример един твърд диск:

Той решава една единствена задача, нали? Твърдият диск решава задачата за съхранение на данни. Той не охлажда компютъра, не издава звуци, няма изчислителна сила и не се ползва като клавиатура. Той е свързан с компютъра само с 2 кабела, т.е. има прост интерфейс за достъп и не е обвързан с другите периферни устройства. Твърдият диск работи самостоятелно и другите устройства не се интересуват от това точно как работи. Централния процесор му казва "чети" и той чете, след това му казва "пиши" и той пише. Как точно го прави е скрито вътре в него. Различните модели могат да работят по различен начин, но това си е техен проблем. Виждате, че един твърд диск притежава strong cohesion, loose coupling, добра абстракция и добра капсулация. Така трябва да реализирате и вашите класове – да вършат една задача, да я вършат добре, да се обвързват минимално с другите класове (или въобще да не се обвързват, когато е възможно), да имат ясен интерфейс и да добра абстракция и да скриват детайлите за вътрешната си работа.

Ето един друг пример: Представете си какво щеше да стане, ако на дънната платка на компютъра бяха запоени процесорът, твърдият диск, CD-ROM устройството и клавиатурата. Това означава, че като ви се повреди някой клавиш от клавиатурата, ще трябва да изхвърлите на боклука целия компютър. Виждате, че при tight coupling и weak cohesion хардуерът не може да работи добре. Същото се отнася и за софтуера.

Обектно-ориентирано моделиране (ООМ)

Нека приемем, че имаме да решаваме определен проблем или задача. Този проблем идва обикновено от реалния свят. Той съществува в дадена реалност, която ще наричаме заобикаляща го среда.

Обектно-ориентираното моделиране (ООМ) е процес, свързан с ООП, при който се изваждат всички обекти, свързани с проблема, който решаваме (създава се модел). Изваждат се само тези техни характеристики, които са свързани с решаването на конкретния проблем. Останалите се игнорират. Така вече си създаваме нова реалност, която е опростена версия

на оригиналната (неин модел), и то такава, че ни позволява да си решим проблема или задачата.

Например, ако моделираме система за продажба на билети, за един пътник важни характеристики биха могли да бъдат неговото име, неговата възраст, дали ползва намаление и дали е мъж или жена (ако продаваме спални места). Пътникът има много други характеристики, които не ни интересуват, примерно какъв цвят са му очите, кой номер обувки носи, какви книги харесва или каква бира харесва.

При моделирането се създава опростен модел на реалността с цел решаване на конкретната задача. При обектно-ориентираното моделиране моделът се прави със средствата на ООП: чрез класове, атрибути на класовете, методи в класовете, обекти, взаимоотношения между класовете и т.н. Нека разгледаме този процес в детайли.

Стъпки при обектно-ориентираното моделиране

Обектно-ориентираното моделиране обикновено се извършва в следните стъпки:

- Идентификация на класовете.
- Идентификация на атрибутите на класовете.
- Идентификация на операциите върху класовете.
- Идентификация на връзките между класовете.

Ще разгледаме кратък пример, с който ще ви покажем как могат да се приложат тези стъпки.

Идентификация на класовете

Нека имаме следната извадка от заданието за дадена система:

> На потребителя трябва да му е позволено да описва всеки продукт по основните му характеристики, включващи име и номер на продукта. Ако бар-кодът не съвпада с продукта, тогава трябва да бъде генерирана грешка на екрана за съобщения. Трябва да има дневен отчет за всички транзакции, специфицирани в секция 9.

Ето как идентифицираме ключовите понятия:

> На **потребителя** трябва да му е позволено да описва всеки **продукт** по основните му **характеристики**, включващи **име** и **номер на продукта**. Ако **бар-кодът** не съвпада с продукта, тогава трябва да бъде генерирана **грешка** на **екрана за съобщения**. Трябва да има **дневен отчет** за всички **транзакции**, специфицирани в секция 9.

Току-що идентифицирахме класовете, които ще ни трябват. Имената на класовете са съществителните имена в текста, най-често нарицателни в единствено число, например **Студент**, **Съобщение**, **Лъв**. Избягвайте имена, които не идват от текста, примерно: **СтраненКлас**, **АдресКойтоИмаСтудент**.

Понякога е трудно да се прецени дали някой предмет или явление от реалния свят трябва да бъде клас. Например адресът може да е клас **Address** или символен низ. Колкото по-добре проучим проблема, толкова по-лесно ще решим кое трябва да е клас. Когато даден клас стане прекалено голям и сложен, той трябва да се декомпозира на няколко по-малки класове.

Идентификация на атрибутите на класовете

Класовете имат атрибути (характеристики), например: класът **Student** има име, учебно заведение и списък от курсове. Не всички характеристики са важни за софтуерната система. Например: за класа **Student** цвета на очите е несъществена характеристика. Само съществените характеристики трябва да бъдат моделирани.

Идентификация на операциите върху класовете

Всеки клас трябва да има ясно дефинирани отговорности – какви обекти или процеси от реалния свят представя, какви задачи изпълнява. Всяко действие в програмата се извършва от един или няколко метода в някой клас. Действията се моделират с операции (методи).

За имената на методите се използват глагол + съществително. Примери: **PrintReport()**, **ConnectToDatabase()**. Не може веднага да се дефинират всички методи на даден клас. Дефинираме първо най-важните методи – тези, които реализират основните отговорности на класа. С времето се появяват още допълнителни методи.

Идентификация на връзките между класовете

Ако един студент е от определен факултет и за задачата, която решаваме, това е важно, тогава студент и факултет са свързани. Тоест класът **Факултет** има списък от **Студенти**. Тези връзки наричаме още асоциации (спомнете си секцията "клас-диаграми").

Нотацията UML

UML (Unified Modelling Language) бе споменат в секцията за наследяване. Там разгледахме клас-диаграмите. UML нотацията дефинира още няколко вида диаграми. Нека разгледаме накратко някои от тях.

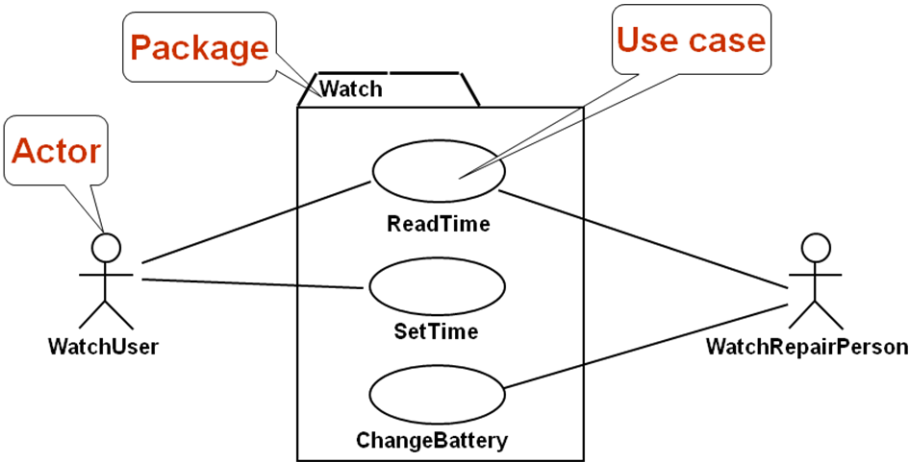
Use case диаграми (случаи на употреба)

Използват се при извличане на изискванията за описание на възможните действия. **Актьорите (actors)** представят роли (типове потребители).

Случаите на употреба (use cases) описват взаимодействие между актьорите и системата. Use case моделът е група use cases – предоставя пълно описание на функционалността на системата.

Use case диаграми – пример

Ето как изглежда една sequence диаграма:



Актьорът е някой, който взаимодейства със системата (потребител, външна система или примерно външната среда). Актьорът има уникално име и евентуално описание.

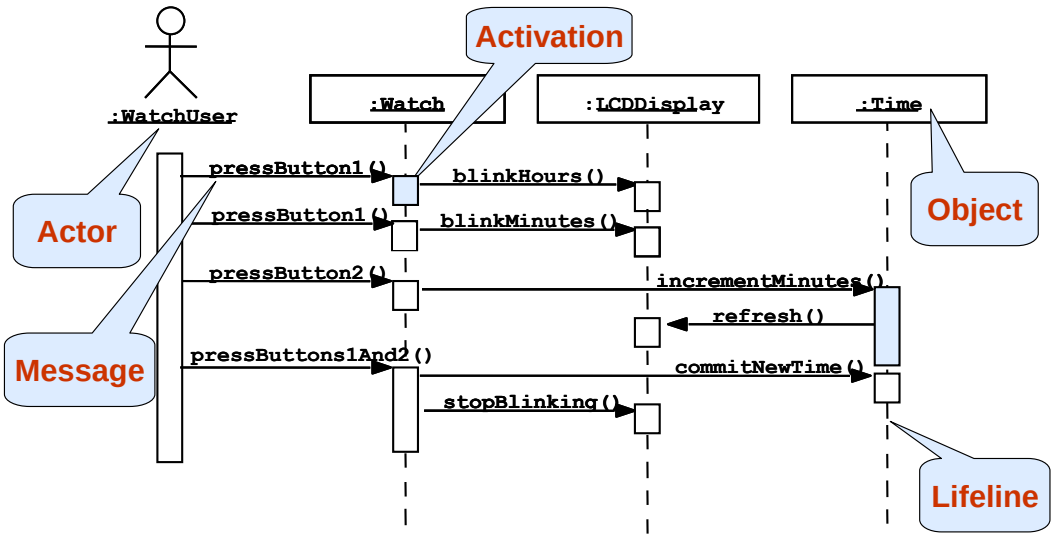
Един use case описва една от функционалностите на системата. Той има уникално име и е свързан с актьори. Може да има входни и изходни условия. Най-често съдържа поток от действия (процес). Може да има и други изисквания.

Sequence диаграми

Използват се при моделиране на изискванията за описание на процеси. За по-добро описание на use case сценариите. Позволяват описание на допълнителни участници в процесите. Използват се при дизайна за описание на системните интерфейси.

Sequence диаграми – пример

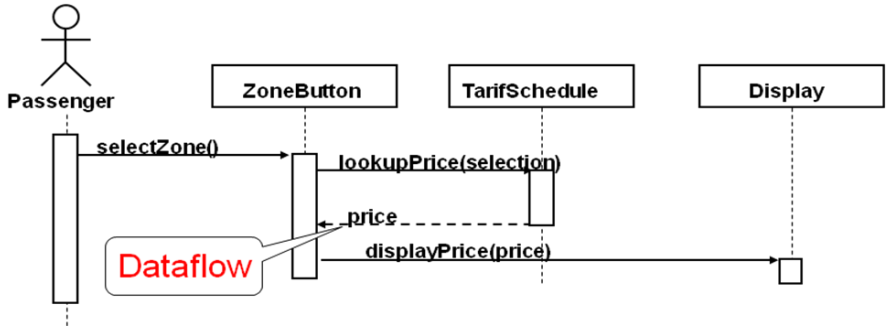
Ето как изглежда една sequence диаграма:



Класовете се представят с колони. **Съобщенията (действията)** се представят чрез стрелки. **Участниците** се представят с широки правоъгълници. **Състоянията** се представят с пунктирана линии.

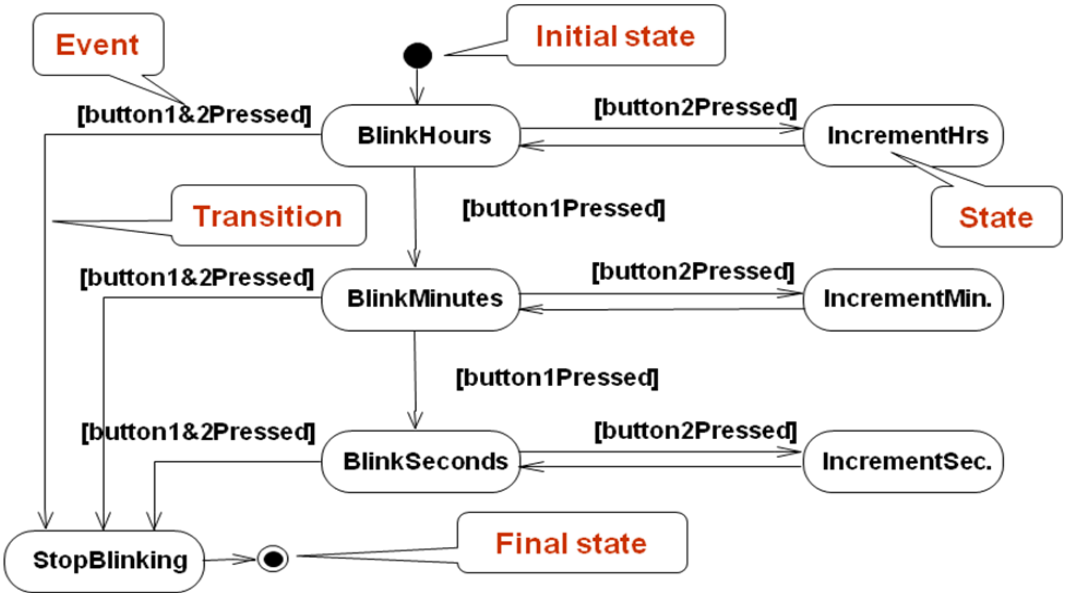
Съобщения – пример

Посоката на стрелката определя изпращача и получателя на съобщението. Хоризонталните прекъснати линии изобразяват потока на данните:



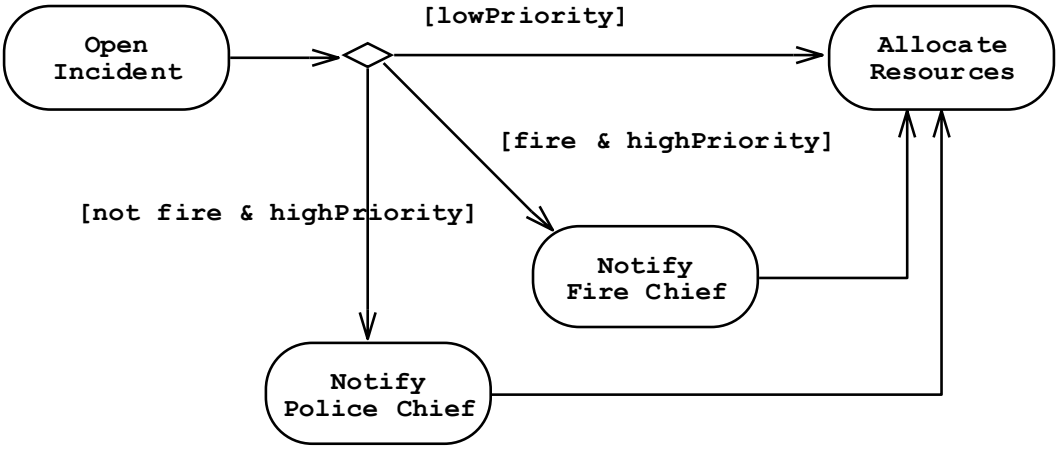
Statechart диаграми

Statechart диаграмите описват възможните състояния на даден процес и възможните преходи между тях. Представляват краен автомат:



Activity диаграми

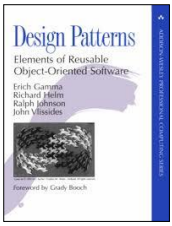
Представляват специален тип statechart диаграми, при които състоянията са действия. Показват потока на действията в системата:



Шаблони за дизайн

Достатъчно време след появата на обектно-ориентираната парадигма се оказва, че съществуват множество ситуации, които се появяват често при писането на софтуер. Например клас, който трябва да има само една инстанция в рамките на цялото приложение.

Появяват се **шаблоните за дизайн (design patterns)** – популярни решения на често срещани проблеми от обектно-ориентираното моделиране. Част от тях са най-добре обобщени в едноименната книга на Ерих Гама "Design Patterns: Elements of Reusable Object Oriented Software" (ISBN 0-201-63361-2).



Това е една от малкото книги на компютърна тематика, които остават актуални 15 години след издаването си. Шаблоните за дизайн допълват основните принципи на ООП с допълнителни добре известни решения на добре известни проблеми. Добро място за започване на разучаването им е статията за тях в Уикипедия: http://en.wikipedia.org/wiki/Design_pattern_(computer_science).

Шаблонът Singleton

Това е най-популярният и използван шаблон. Позволява на определен клас да има само една инстанция и дефинира откъде да се вземе тази инстанция. Типични примери са класове, които дефинират връзка към единствени неща (виртуалната машина, операционна система, мениджър на прозорците при графично приложение, файлова система), както и класовете от следващия шаблон (factory).

Шаблонът Singleton – пример

Ето примерна имплементация на шаблона Singleton:

Singleton.java

```
package introjavabook;

public class Singleton {

   // Single instance
   private static Singleton instance;

   // Initialize the single instance
   static {
      instance = new Singleton();
   }

   // The method for taking the single instance
   public static Singleton getInstance() {
      return instance;
   }

   // Private constructor – protects direct instantialion
   private Singleton(){}
```

```
}
```

Имаме скрит конструктор, за да ограничим инстанциите. Имаме статична променлива, която държи единствената инстанция. Инициализираме я еднократно в статичния конструктор на класа. Методът за вземане на инстанцията най-често се казва **getInstance()**.

Шаблонът може да претърпи много оптимизации, например мързеливо инициализиране на единствената променлива за спестяване на памет, но това е класическата му форма.

Шаблонът Factory Method

Factory method е друг много разпространен шаблон. Той е предназначен да "произвежда" обекти. Инстанцирането на определен обект не се извършва директно, а се прави от factory метода. Това позволява на factory метода да реши коя конкретна инстанция да създаде. Решението може да зависи от външната среда, от параметър или от някаква системна настройка.

Шаблонът Factory Method – пример

Ще извадим един пример директно от Java платформата:

java.lang.Integer

```
public final class Integer
      extends Number
      implements Comparable<Integer> {

   // ...

   public static Integer valueOf(String s)
         throws NumberFormatException {
      return new Integer(parseInt(s, 10));
   }

   // ...

}
```

Методът **valueOf(String)** произвежда инстанция (число) на базата на символен низ. Има и параметър, който се подразбира - числото 10, което указва в каква бройна система се очаква да е числото в символния низ.

Шаблонът Factory Method – втори пример

Примерът отново е от стандартната библиотека на Java:

java.util.Calendar

```
package java.util;


public abstract class Calendar implements Serializable,/*...*/ {

   // ...

   public static Calendar getInstance() {
      Calendar cal = createCalendar(
         TimeZone.getDefaultRef(),
         Locale.getDefault());
      cal.sharedZone = true;
      return cal;
   }

   private static Calendar createCalendar(
           TimeZone zone, Locale aLocale) {

      // If the specified locale is a Thai locale,
      // returns a BuddhistCalendar instance.
      if ("th".equals(aLocale.getLanguage())
           && ("TH".equals(aLocale.getCountry()))) {
         return new sun.util.BuddhistCalendar(zone, aLocale);
      } else if ("JP".equals(aLocale.getVariant())
           && "JP".equals(aLocale.getCountry())
           && "ja".equals(aLocale.getLanguage())) {
         return new JapaneseImperialCalendar(zone, aLocale);
      }

      // else create the default calendar
      return new GregorianCalendar(zone, aLocale);
   }
   // ...
}
```

Можем да приемем, че и двата метода са factory методи. Методът **getInstance()** съобразява създаването на инстанцията с околната среда - локала (**Locale**) и часовата зона. После използва друг factory метод, който да създаде реално инстанцията.

Методът **createCalendar()** връща инстанция на класа, съобразена с локала (**Locale**) и часовата зона, подадени като параметри. На базата на тях се връща Будистки, Японски или Григориански календар.

Други шаблони

Съществуват десетки други добре известни шаблони за дизайн, но няма да се спираме подробно на тях. По-любознателните читатели могат да потърсят за "Design Patterns" в Интернет и да разберат за какво случат и как се използват шаблони като: abstract factory, prototype, adapter, composite, façade, command, iterator, observer и много други. Ако продължите да се занимавате с Java по-сериозно, ще се убедите, че цялата стандартна библиотека (Java API) е конструирана върху принципите на ООП и използва много активно класическите шаблони за дизайн.

Упражнения

1. Дефинирайте клас **`Human`** със свойства "собствено име" и "фамилно име". Дефинирайте клас **`Student`**, наследяващ **`Human`**, който има свойство "оценка". Дефинирайте клас **`Worker`**, наследяващ **`Human`**, със свойства "надница" и "изработени часове". Имплементирайте и метод "изчисли надница за 1 час", който смята колко получава работникът за 1 час работа, на базата на надницата и изработените часове. Напишете съответните конструктори и методи за достъп до полетата (свойства).
2. Инициализирайте масив от 10 студента и ги сортирайте по оценка в нарастващ ред. Използвайте Java интерфейса **`java.lang.Comparable`**.
3. Инициализирайте масив от 10 работника и ги сортирайте по заплата в намаляващ ред.
4. Дефинирайте клас **`Shape`** със само един метод **`calculateSurface()`** и полета **`width`** и **`height`**. Дефинирайте два нови класа за триъгълник и правоъгълник, които имплементират споменатия виртуален метод. Този метод трябва да връща площта на правоъгълника (**`height*width`**) и триъгълника (**`height*width/2`**). Дефинирайте клас за кръг с подходящ конструктор, при когото при инициализация и двете полета (**`height`** и **`width`**) са с еднаква стойност (радиуса), и имплементирайте виртуалния метод за изчисляване на площта. Направете масив от различни фигури и сметнете площта на всичките в друг масив.
5. Имплементирайте следните обекти: куче (**`Dog`**), жаба (**`Frog`**), котка (**`Cat`**), котенце (**`Kitten`**), котарак (**`Tomcat`**). Всички те са животни (**`Animal`**). Животните се характеризират с възраст (**`age`**), име (**`name`**) и пол (**`gender`**). Всяко животно издава звук (виртуален метод на **`Animal`**). Направете масив от различни животни и за всяко изписвайте на конзолата името, възрастта и звука, който издава.
6. Изтеглете си някакъв инструмент за работа с UML и негова помощ генерирайте клас диаграма на класовете от предходната задача.
7. Прочетете за шаблона "abstract factory" и го имплементирайте.

Решения и упътвания

1. Задачата е тривиална. Просто следвайте условието и напишете кода.
2. Имплементирайте **`Comparable`** в **`Student`** и оттам просто сортирайте списък от **`Comparable`**. Можете да използвате и **`java.util.Arrays. sort(Object[])`**.
3. Задачата е като предната.
4. Имплементирайте класовете както са описани в условието на задачата. Тествайте решението си.
5. Изписването на информацията можете да го имплементирате във виртуалния метод **`java.lang.Object.toString()`**. За да принтирате съдържанието на целия масив, можете да ползвате статичния метод **`java.util.Arrays.toString(Object[])`**, който ще използва предефинирания от вас **`toString()`**.
6. Можете да намерите списък с UML инструменти от следния адрес: http://en.wikipedia.org/wiki/List_of_UML_tools.
7. Можете да прочетете за шаблона "abstract factory" от Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern.

Глава 21. Качествен програмен код

Автор

Михаил Стойнов

Светлин Наков

Николай Василев

В тази тема...

В настоящата тема ще разгледаме основните правила за писане на качествен програмен код. Ще бъде обърнато внимание на именуването на елементите от програмата (променливи, методи, класове и други), правилата за форматиране и подреждане на кода, добрите практики за изграждане на висококачествени методи и принципите за качествена документация на кода. Ще бъдат дадени много примери за качествен и некачествен код. Ще бъдат описани и официалните конвенции от Sun за писане на Java, както и JavaBeans спецификацията. В процеса на работа ще бъде обяснено как да се използва средата за програмиране, за да се автоматизират някои операции като форматиране и преработка на кода.

Какво е качествен програмен код?

Качеството на една програма има два аспекта – качеството, измерено през призмата на потребителя (наречено **външно качество**), и от гледна точна на вътрешната организация (наречено **вътрешно качество**).

Външното качество зависи от това колко коректно работи тази програма. Зависи също от това колко е интуитивен и ползваем е потребителският интерфейс. Зависи и от производителността (колко бързо се справя с поставените задачи).

Вътрешното качество е свързано с това колко добре е построена тази програма. То зависи от архитектурата и дизайна (дали са достатъчно изчистени и подходящи). Зависи от това колко лесно е да се направи промяна или добавяне на нова функционалност (леснота за поддръжка). Зависи и от простотата на реализацията и четимостта на кода. Вътрешното качество е свързано най-вече с кода на програмата.

Характеристики за качество на кода

Качествен програмен код е такъв, който се чете и разбира лесно. Той трябва да е коректен, да има добро форматиране, което консистентно се прилага навсякъде. На всички нива (модули, класове, методи) трябва да има висока свързаност на отговорностите (strong cohesion) и функционална независимост (loose coupling). Подходящо и консистентно именуване на класовете, методите, променливите и останалите елементи също е задължително условие. Кодът трябва да има и добра документация, вградена в него самия.

Защо трябва да пишем качествено?

Нека погледнем следния код:

```
public static void main(String... args) {
   int i, j, w;
   final int value=010; i=5;
   switch(value){case 10:w=5;case 9:i=0;default:;
      System.out.print("4 ");break;
        case 8:System.out.print("9 ");
      System.out.print("9 ");{
         System.out.print("9 ");}
      for(int k = 0;k < i;k++,System.out.print(k-'f'));}{
         System.out.println("Cycle");}
};
```

Можете ли да кажете дали този код се компилира без грешки? Можете ли да кажете какво прави само като го гледате? Можете ли да добавите нова функционалност и да сте сигурни, че няма да счупите нищо старо? Можете ли да кажете за какво служи променливата **k** или променливата **w**?

В Eclipse има опция за пренареждане на код. Ако горният код бъде сложен в Eclipse и се извика тази опция (клавишна комбинация [Ctrl+Shift+F]), кодът ще бъде преформатиран и ще изглежда съвсем различно. Въпреки това все още няма да е ясно за какво служат променливите, но поне ще е ясно кой блок с код къде завършва.

Ако всички пишеха код както в примера, нямаше да е възможно реализирането на големи и сериозни софтуерни проекти, защото те се пишат от големи екипи от софтуерни инженери. Ако кодът на всички е като в примера по-горе, никой няма да е в състояние да разбере как работи (и дали работи) кодът на другите от екипа, а с голяма вероятност никой няма да си разбира и собствения код.

С времето в професията на програмистите се е натрупал сериозен опит и добри практики за писане на качествен програмен код, за да е възможно всеки да разбере кода на колегите си и да може да го променя и дописва. Тези практики представляват множество от препоръки и правила за

форматиране на кода, за именуване на идентификаторите и за правилно структуриране на програмата, които правят писането на софтуер по-лесно. Качественият и консистентен код помага най-вече за поддръжката и лесната промяна. Качественият код е гъвкав и стабилен. Той се чете и разбира лесно от всички. Ясно е какво прави от пръв поглед, поради това е самодокументиращ се. Качественият код е интуитивен – ако не го познавате има голяма вероятност да познаете какво прави само с един бърз поглед. Качественият код е удобен за преизползване, защото прави само едно нещо (strong cohesion), но го прави добре, като разчита на минимален брой други компоненти (loose coupling) и ги използва само през публичните им интерфейси. Качественият код спестява време и труд и прави написания софтуер по-ценен.

Код-конвенции

Преди да продължим с препоръките за писане на качествен програмен код ще поговорим малко за код-конвенции. **Код-конвенция** е група правила за писане на код, използвана в рамките на даден проект или организация. Те могат да включват правила за именуване, форматиране и логическа подредба. Едно такова правило например може да препоръчва класовете да започват с главна буква, а променливите – с малка. Друго правило може да твърди, че къдравата скоба за нов блок с програмни конструкции се слага на същия ред, а не на нов ред.

Конвенциите са започнали да се появяват в големи и сериозни проекти, в които голям брой програмисти са пишели със собствен стил и всеки от тях е спазвал собствени (ако въобще е спазвал някакви) правила. Това е правело кода по-трудно четим и е принудило ръководителите на проектите да въведат писани правила. По-късно най-добрите код конвенции са придобили популярност и са станали де факто стандарт.

Sun, компанията, която стои зад езика и платформата Java, публикува официално своята код-конвенция за писане на Java още през далечната 1999 година. От тогава тази код конвенция е добила голяма популярност и е широко разпространена. Правилата за именуване на идентификаторите и за форматиране на кода, които ще дадем в тази тема, са в синхрон с код конвенцията на Sun.

Неконсистентното използване на една конвенция е по-лошо и по-опасно от липсата на конвенция въобще.

Именуване на идентификаторите

Идентификатори са имената на класове, интерфейси, изброими типове, анотации, методи и променливи. В Java и в много други езици имената на идентификаторите се избират от разработчика. Имената не трябва да бъдат случайни. Те трябва да са съставени така, че да носят полезна информация

за какво служат и каква точно роля изпълняват в съответния код. Така кодът става по-лесно четим.

Едно от най-основните правила е, винаги да се използва английски език. Помислете само ако някой виетнамец използва виетнамски език, за да си кръщава променливите и методите. Какво ще разберете, ако четете неговия код? Ами какво ще разбере виетнамецът, ако вие сте ползвали български и след това той се наложи да допише вашия код. Единственият език, който всички програмисти владеят, е английският.

Английският език е де факто стандарт при писането на софтуер. Винаги използвайте английски език за имената на идентификаторите в сорс кода (променливи, методи, класове и т.н.).

Нека сега разгледаме как да подберем подходящите идентификатори в различните случаи.

Имена на класове, интерфейси и други типове

От главата "Принципи на обектно-ориентираното програмиране" знаем, че класовете описват обекти от реалния свят. Имената на класовете трябва да са съставени от съществително име (нарицателно или собствено) и от едно или няколко прилагателни (преди или след съществителното). Например класът описващ Африканския лъв ще се казва **`AfricanLion`**. Тази нотация на именуване се нарича **Pascal Case** – първата буква на всяка дума от името е главна, а останалите са малки. Така по-лесно се чете (забележете разликата между **`concurrenthashmap`** срещу **`ConcurrentHashMap`**).

Да дадем още няколко примера. Трябва да напишем клас, който намира прости числа в даден интервал. Добро име за този клас е **`PrimeNumbers`** или **`PrimeNumbersFinder`** или **`PrimeNumbersScanner`**. Лоши имена биха могли да бъдат **`FindPrimeNumber`** (не трябва да ползваме глагол за име на клас) или **`Numbers`** (не става ясни какви числа и какво ги правим) или Prime (не трябва да името на клас да е прилагателно).

Изключително лошо име на клас е **`Problem12`**. Някои начинаещи програмисти дават такова име за решението на задача 12 от упражненията. Това е изключително грешно! Какво ще ви говори името **`Problem12`** след 1 седмица или след 1 месец? Ако задачата търси път в лабиринт, дайте и име **`PathInLabyrinth`**. След 3 месеца може да имате подобна задача и да трябва да намерите задачата за лабиринта. Как ще я намерите, ако не сте й дали подходящо име? Не давайте име, което съдържа числа – това е индикация за лошо именуване.

Името на класа трябва да описва за какво служи този клас. Решението на задача 12 от упражненията не трябва да се казва `Problem12` или `Zad12`. Това е груба грешка!

Избягвайте съкращения

Съкращения трябва се избягват, защото могат да бъдат объркващи. Например за какво ви говори името на клас **GrBxPnl**? Не е ли по-ясно, ако името е **GroupBoxPanel**? Изключения се правят за акроними, които са по-популярни от пълната си форма, например HTML или URL. Например името **HTMLParser** е препоръчително пред **HyperTextMarkupLanguageParser**.

Колко да са дълги имената на класовете?

Имената на класовете не трябва да надвишават в общия случай 20 символа, но понякога това правило не се спазва, защото се налага да се опише обект от реалността, който се състои от няколко дълги думички. В стандартните Java библиотеки има класове с дължина 40 символа, като примерно класовете **ContentHandlerAlreadyRegisteredException** и **SQLIntegrityConstraintViolationException**. Въпреки дължината е ясно за какво служат и двата класа. По тази причината препоръката за дължина до 20 символа, е само ориентировъчна, а не задължителна. Ако може едно име да е по-кратко и също толкова ясно, колкото дадено по-дълго име, предпочитайте по-краткото.

Имена на интерфейси и други типове

Имената на интерфейсите, изброимите типове (enums) и анотациите трябва да следват същата конвенция, както имената на класовете: изписват се в Pascal Case и се състоят от съществително и евентуално прилагателни.

В Java има още една нотация за имена интерфейси: да завършват на **able**: **Runnable**, **Serializable**, **Cloneable**. Това са интерфейси, които най-често добавят допълнителна роля към основната роля на един обект. В Java повечето интерфейси не следват тази нотация, например интерфейсите **Map** и **Set**.

Имена на методи

В имената на методите отново всяка отделна дума трябва да е с главна буква, но за разлика от Pascal Case, тук първата буква е малка. Тази нотация се нарича **camelCase**.

Имената на методите трябва да се съставят по схемата <глагол> + <обект>, например **printReport()**, **loadSettings()** или **setUserName()**. Обектът може да е съществително или да е съставен от съществително и прилагателно, например **showAnswer()**, **connectToRandomTorrentServer()** или **findMaxValue()**.

Като примери за лоши имена на методи можем да дадем следните: **doWork()** (не става ясно каква точно работа върши), **printer()** (няма глагол), **find2()** (ами защо не е **find7()?**), **chkErr()** (не се препоръчват съкращения), **nextPosition()** (няма глагол).

Понякога единични глаголи са също добро име за метод, стига да става ясно какво прави съответния метод и върху какви обекти оперира. Например ако имаме клас **Task**, методите **start()**, **stop()** и **cancel()** са с добри имена, защото става ясно, че стартират, спират или оттеглят изпълнението на задачата, в текущия обект (**this**). В други случаи единичния глагол е грешно име, примерно в клас с име **Utils** методи с имена **evaluate()**, **create()** или **stop()** са неадекватни.

Методи, които връщат стойност

Имената на методите, които връщат стойност, трябва да описват връщаната стойност, например **getNumberOfProcessors()**, **findMinPath()**, **getPrice()**, **getRowsCount()**, **createNewInstance()**.

Примери за лоши имена на методи, които връщат стойност (функции) са следните: **showReport()** (не става ясно какво връща методът), **value()** (трябва да е **getValue()** или **hasValue()**), **student()** (няма глагол), **empty()** (трябва да е **isEmpty()**).

Свързаност на отговорностите и именуване

Името трябва да описва всичко, което методът извършва. Ако не може да се намери подходящо име, значи няма силна свързаност на отговорностите (strong cohesion), т.е. методът върши много неща едновременно и трябва да се раздели на няколко отделни метода.

Ето един пример: имаме метод, който праща e-mail, печата отчет на принтер и изчислява разстояние между точки в тримерното евклидово пространство. Какво име ще му дадем? Може би ще го кръстим **sendEmailAndPrintReportAndCalc3DDistance()**? Очевидно е, че нещо не е наред с този метод – трябва да преработим кода вместо да се мъчим да дадем добро име. Още по-лошо е, ако дадем грешно име, примерно **sendEmail()**. Така подвеждаме всички останали програмисти, че този метод праща поща, а той всъщност прави много други неща.

Даването на заблуждаващо име за метод е по-лошо дори от това да го кръстим `method1()`. Например ако един метод изчислява косинус, а ние му дадем за име `sqrt()`, ще си навлечем яростта на всички колеги, които се опитват да ползват нашия код.

Именуване на методи – още препоръки

Лоша практика е за имена на методи да се използват безлични и общи думички, например **handleStuff()** и **processData()**. Какво вършат тези методи според вас? Става ли ясно?

Не се препоръчва да се използват цифри в името, например **readProfile1()** и **readProfile2()**. При такива имена веднага възниква въпросът каква е разликата между методите и защо нямаме **readProfile3**.

Дължината на името трябва да е толкова дълга, колкото е необходимо. Нормалното име на метод е в рамките на 9-15 символа. Ако името е прекалено дълго, ней-вероятно имаме лоша кохезия. Това правило не е твърдо и служи само за ориентир.

Имената на методите трябва да са на английски език. Вече обяснихме защо – никой не иска да чете код писан от чужденци с имена на методите на техния си език.

Консистентно трябва да се именуват методи, които извършват противоположни операции: **open()** и **close()**, **read()** и **write()**. Лош пример би бил: **openFile()** и **_descriptor_close()**.

Използвайте конвенция за честите операции. Например за четене и писане можем да ползваме **getXXX()** и **setXXX()**: **getName()**, **getAge()**, **setName()**, **setAge()**. Спазвайте тази конвенция навсякъде.

Модификатори

В света на Java има още една изключително популярна спецификация. Нарича се JavaBeans (пише се слято). JavaBeans е група от правила и интерфейси за писане на преизползваеми компоненти. Нека разгледаме правилата, които се отнасят до методите. По-късно ще опишем по-подробно тази спецификация.

JavaBeans препоръчва, всяка една член-променлива да бъде скрита (**private**), а достъпът до нея да се осъществява от специални методи наречени **модификатори** (getters and setters, accessor methods).

Имената на тези методи са изведени от името на променливата. За всяка член-променлива има два метода – един за четене и един за писане. На метода за четене името му е **get** + <името на променливата>, на метода за писане – **set** + <името на променливата>. Името на променливата и в двата случая от CamelCase става в Pascal case.

Например ако името на променливата е **numberOfProcessors**, то модификаторите (accessors) ще са с имена **getNumberOfProcessors()** и **setNumberOfProcessors()**.

Ето един пример за използване на JavaBeans конвенцията за капсулиране на достъпа до член-променливи:

Person.java

```
package introjavabook;
```

```
public class Person {

   private String name;
   private String age;
   private int numberOfChildren;
   private boolean male;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getAge() {
      return age;
   }

   public void setAge(String age) {
      this.age = age;
   }

   public int getNumberOfChildren() {
      return numberOfChildren;
   }

   public void setNumberOfChildren(int numberOfChildren) {
      this.numberOfChildren = numberOfChildren;
   }

   public boolean isMale() {
      return male;
   }

   public void setMale(boolean male) {
      this.male = male;
   }
}
```

Забележете, че член-променливата е **private**, а модификаторите – **public**. Забележете, че методът за четене (getter) на член-променливата **male** е **isMale()** вместо **getMale()**. JavaBeans спецификацията повелява думичката **is** да се използва вместо **get** за булевите член-променливи.

Имена на променливи

Имената на променливите (променливи използвани в метод) и член-променливите (променливи използвани в клас) според Sun конвенцията трябва да спазват camelCase нотацията.

Променливите трябва да имат добро име. Добро име е такова, което ясно и точно описва обекта, който променливата съдържа. Например добри имена на променливи са **account**, **blockSize** и **customerDiscount**. Лоши имена са: **r18pq**, **__hip**, **rcfd**, **val1**, **val2**.

Името трябва да адресира проблема, който решава променливата. Тя трябва да отговаря на въпроса "какво", а не "как". В този смисъл добри имена са **employeeSalary**, **employees**. Лоши имена са, несвързаните с решавания проблем имена **myArray**, **customerFile**, **customerHashTable**.

Оптималната дължина на името на променлива е от 10 до 16 символа. Изборът на дължината на името зависи от обхвата – променливите с по-голям обхват и по-дълъг живот имат по-дълго и описателно име:

```
protected Account[] customerAccounts;
```

Променливите с малък обхват и кратък живот могат да са по-кратки:

```
for (int i=0; i<customers.Length; i++) { … }
```

Имената на променливите трябва да са разбираеми без предварителна подготовка. Поради тази причина не е добра идея да се премахват гласните от името на променливата с цел съкращение – **btnDfltSvRzlts** не е много разбираемо име.

Най-важното е, че каквито и правила да бъдат изградени за именуване на променливите, те трябва да бъдат консистентно прилагани навсякъде из кода, в рамките на всички модули на целия проект и от всички членове на екипа. Неконсистентно прилаганото правило е по-опасно от липсата на правило въобще.

Имена на константи

В Java константите са статични непроменими променливи и се дефинират по следния начин:

```
public class ThreadPool {
   public static final int <variable name> = <value>;
}
```

Имената на константите трябва да се изписват изцяло с главни букви с долна черта между думите. Пример:

```
public class ThreadPool {
   public static final int MAX_POOL_SIZE = 16;
}
```

Имената на константите точно и ясно трябва да описват смисъла на даденото число, стринг или друга стойност, а не самата стойност. Например, ако една константа се казва **number314159**, тя е безполезна.

Именуване на специфични типове данни

Имената на променливи, използвани за броячи, е хубаво да включват в името си дума, която указва това, например **usersCount**, **rolesCount**, **filesCount**.

Променливи, които се използват за описване на състояние на даден обект, трябва да бъдат именувани подходящо. Ето няколко примера: **ThreadState**, **TransactionState**.

Временните променливи най-често са с безлични имена (което указва, че са временни променливи, т.е. имат много кратък живот). Добри примери са **index**, **value**, **count**. Неподходящи имена са **a**, **aa**, **tmpvar1**, **tmpvar2**.

Имената на булевите променливи трябва да дават предпоставка за истина или лъжа. Например: **canRead**, **available**, **isOpen**, **valid**. Примери за неадекватни имена на булеви променливи са: **student**, **read**, **reader**.

Именуване с префикси или суфикси

В по-старите езици (например C) съществуват префиксни или суфиксни нотации за именуване. Много популярна в продължение на много години е била Унгарската нотация. Унгарската нотация е префиксна конвенция за именуване, чрез която всяка променлива получава префикс, който обозначава типа ѝ или предназначението ѝ. Например в Win32 API името **lpcstrUserName** би означавало променлива, която представлява указател към масив от символи, който завършва с 0 и се интерпретира като стринг.

В Java подобни конвенции не са придобили популярност, защото средите за Java показват типа на всяка променлива. Изключение донякъде правят графични библиотеки като Swing и AWT.

Форматиране на кода

Форматирането, заедно с именуването, е едно от основните изисквания за четим код. Без форматиране, каквито и правила да спазваме за имената и структурирането на кода, кодът няма да се чете лесно.

Целта на доброто форматиране е да направи кода по-ясен и по-лесен за четене. Ако форматирането прави кода по-труден за четене, значи не е добро. Всяко форматиране (отместване, празни редове, подреждане,

подравняване и т.н.) може да донесе както ползи, така и вреди. Важно е форматирането на кода да следва логическата структура на програмата, така че да подпомага четенето и логическото й разбиране.

Форматирането на програмата трябва да разкрива неговата логическа структура. Всички правила за форматиране на кода имат една и съща цел – подобряване на четимостта на кода чрез разкриване на логическата му структура.

В Eclipse кодът може да се форматира автоматично с клавишната комбинация [Ctrl+Shift+F]. Могат да бъдат зададени различни стандарти за форматиране на код – Sun конвенцията, стандартът на Eclipse, както и потребителски дефинирани стандарти.

Сега ще разгледаме правилата за форматиране от код-конвенцията на Sun за Java.

Правила за форматиране на метод

Съгласно конвенцията за писане на код, препоръчана от Sun, е добре да се спазват някои правила за форматиране на кода, при декларирането на методи.

Форматиране на множество декларации на методи

Когато в един клас имаме повече от един метод, трябва да разделяме декларациите им с един празен ред:

IndentationExample.java

```
public class IndentationExample {

   public static void doSth1() {
      // ...
   }// Follows one blank line

   public static void doSth2() {
      // ...
   }
}
```

Форматиране на декларацията на метод

Декларацията на метода ни е на първо място на реда, на който се намира (т.е. няма никакъв друг код пред нея), отместена с един знак за табулация по-навътре, спрямо началото на декларацията на класа, в който е деклариран метода:

```
public class IndentationExample {

   // The following method definition is indented
   public static void doSth() {
      // ... Code ...
   }
}
```

Как да поставяме кръгли скоби?

В конвенцията за писане на код, на Sun, се препоръчва, между ключова дума, като например – **for**, **while**, **if**, **switch**... и отваряща скоба да поставяме интервал:

```
while (true) {
   // ... Code ...
}
```

Това се прави с цел да се различават по-лесно ключовите думи и имената на методите.

В този ред на мисли, между името на метода и отварящата кръгла скоба – "(", **НЕ трябва** да има невидими символи (интервал, табулация и т.н.):

```
public static void printLogo() {
   // ... Code ...
}
```

Форматиране на списъка с параметри на метод

Когато имаме метод с много параметри, трябва добре да оставяме един интервал разстояние между поредната запетайка и типа на следващия параметър:

```
public void doSth(int param1, int param2, int param3) {}
```

Съответно, същото правило прилагаме, когато извикваме метод с повече от един параметър. Преди аргументите, предшествани от запетайка, поставяме интервал:

```
doSth(1, 2, 3);
```

Форматиране на тялото на метод

Съгласно конвенцията на Sun за писане на код, трябва да поставяме отварящата скоба на тялото на метода "**{**", на същия ред, на който е декларацията на метода. Това правило поражда много спорове с програмисти, които са свикнали да пишат на други езици (например на C#).

Затварящата скоба на тялото на метода "**}**", трябва да се намира на нов ред, точно след края на тялото на метода. Отместването на затварящата скоба, трябва да съвпада с отместването на началото на декларацията на метода. Ето един пример:

```
public static void printLogo() {
   // ... Code ...
}
```

В случай, че тялото на метода е празно, поставяме затварящата скоба на метода непосредствено след отварящата:

```
public static void printLogo() {}
```

Кодът, който се намира в тялото на метода ни, трябва да започва с отместване от една табулация по-навътре, спрямо отместването, от което започва декларацията на метода:

```
public static void printLogo() {
   // ... The code is indented with one tabulator ...
}
```

Правила за форматирането на клас

Когато създаваме класове също е добре да следваме няколко препоръки от Sun за форматиране на кода в класовете.

Място на фигурните скоби на тялото на класа

Отварящата фигурна скоба "**{**" на тялото на класа трябва да е на края на същия ред, на който приключва декларацията на класа.

Затварящата скоба на тялото на класа "**}**" трябва да се намира на същото отместване, като отместването на началото на дефиницията на класа. Ето един пример:

```
public class Boo {
   // ... code ...
}
```

В случай, че тялото на класа е празно, затварящата скоба на класа трябва да бъде поставена непосредствено след отварящата:

```
public class Boo {}
```

Кодът, който се намира в тялото на класа, трябва да започва с отместване от една табулация по-навътре, спрямо отместването, от което започва дефиницията на класа:

```
public class Boo {
   // ... the code is indented with one tabulator ...
}
```

Правила за подредбата на съдържанието на класа

Както знаем, на първия ред се декларира името на класа, предхождано от ключовата дума **class**:

```
// class SomeClassName
```

След това се декларират статичните полета на класа, като първо се декларират тези с модификатор за достъп **public**, след това тези с **protected** и накрая – с **private**:

```
// Class (static) variables
```

След статичните полета на класа, се декларират и нестатичните полета. По подобие на статичните, първо се декларират тези с модификатор за достъп **public**, след това тези с **protected** и накрая – тези с **private**:

```
// Instance variables
```

След нестатичните полета на класа, идва ред на декларацията на конструкторите:

```
// Constructors
```

Най-накрая, след конструкторите, се декларират методите на класа. Препоръчва се да групираме методите по функционалност, вместо по ниво на достъп или област на действие. Например, метод с модификатор за достъп **private**, може да бъде между два метода с модификатори за достъп – **public**. Целта на всичко това е да се улесни четенето и разбирането на кода:

```
// Methods grouped by functionality
```

Подредба на съдържанието на класа – пример

Нека разгледаме един клас, в който съдържанието на класа е подредено по конвенцията:

Dog.java

```
//class statement
public class Dog {
```

```
    // Class (static) variables
    public static final String SPECIES = "Canis Lupus Familiaris";

    // Instance variables
    private String name;

    // Constructors
    public Dog(String name) {
        this.name = name;
    }

    // Methods grouped by functionality

    // -- Getters and setters

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // -- Other methods

    public void breath() {
        // ... code executing the breathing process
    }

    public void bark() {
        System.out.println("wow-wow");
    }
}
```

Съответно методите са групирани в две групи. Едната група е свързана с извличане и модифициране на стойностите на полетата на класа. В другата са методи (в нашия клас само два – **bark()** и **breath()**), които са свързани със спецификата на дейностите, които обектите от класа **Dog** могат да извършват – дишане и лаене.

Правила за форматирането на цикли и условни конструкции

Форматирането на цикли и условни конструкции следва правилата за форматиране на методи и класове. Тялото на условна конструкция или цикъл задължително се поставя в блок, започващ с "{" и завършващ със

"}". Скобата се поставя на същия ред, веднага след условието на цикъла или условната конструкция. Тялото на цикъл или условна конструкция задължително се отмества надясно с една табулация. Ако тялото съдържа само един оператор, също се слагат скоби, макар и компилаторът да не ни задължава да го правим. Ако условието е дълго и не се събира на един ред, се пренася на нов ред с две табулации надясно. Ето пример за коректно форматирани цикъл и условна конструкция:

```
for (List<DictionaryEntry<K, V>> chain :
     this.getHashTableStorage()) {
   if (chain != null) {
      entries.addAll(chain);
   }
}
```

Изключително грешно е да се използва отместване от края на условието на цикъла или условната конструкция като в този пример:

```
for (Student s : students) {
                    System.out.println(s.getName());
                    System.out.println(s.getAge());
                  }
```

Използване на празни редове

Типично за начинаещите програмисти е да поставят безразборно в програмата си празни редове. Наистина, празните редове не пречат, защо да не ги поставяме, където си искаме и защо да ги чистим, ако няма нужда от тях? Причината е много проста: празните редове се използват за разделяне на части от програмата, които не са логическо свързани. Празни редове се поставят за разделяне на методите един от друг, за отделяне на група член-променливи от друга група член-променливи, които имат друга логическа задача, за отделяне на група програмни конструкции от друга група програмни конструкции, които представляват две отделни части на програмата.

Ето един пример, в който празните редове не са използвани правилно и това затруднява четимостта на кода:

```
public static void printList(ArrayList<Integer> list) {
   System.out.print("{ ");
   for (Integer item : list) {
      System.out.print(item);

      System.out.print(" ");
```

```
   }
   System.out.println("}");
}
public static void main(String[] args) {
   ArrayList<Integer> firstList = new ArrayList<Integer>();
   firstList.add(1);

   firstList.add(2);
   firstList.add(3);
   firstList.add(4);
   firstList.add(5);
   System.out.print("firstList = ");
   printList(firstList);
   ArrayList<Integer> secondList = new ArrayList<Integer>();
   secondList.add(2);

   secondList.add(4);
   secondList.add(6);
   System.out.print("secondList = ");
   printList(secondList);
   ArrayList<Integer> unionList = new ArrayList<Integer>();
   unionList.addAll(firstList);
   unionList.removeAll(secondList);

   unionList.addAll(secondList);
   System.out.print("union = ");

   printList(unionList);
}
```

Сами виждате, че празните редове не показват логическата структура на програмата, с което нарушават основното правило за форматиране на кода. Ако преработим програмата, така че да използваме правилно празните редове за отделяне на логически самостоятелните части една от друга, ще получим много по-лесно четим код:

```
public static void printList(ArrayList<Integer> list) {
   System.out.print("{ ");
   for (Integer item : list) {
      System.out.print(item);
      System.out.print(" ");
   }
   System.out.println("}");
}

public static void main(String[] args) {
   ArrayList<Integer> firstList = new ArrayList<Integer>();
```

```
   firstList.add(1);
   firstList.add(2);
   firstList.add(3);
   firstList.add(4);
   firstList.add(5);
   System.out.print("firstList = ");
   printList(firstList);

   ArrayList<Integer> secondList = new ArrayList<Integer>();
   secondList.add(2);
   secondList.add(4);
   secondList.add(6);
   System.out.print("secondList = ");
   printList(secondList);

   ArrayList<Integer> unionList = new ArrayList<Integer>();
   unionList.addAll(firstList);
   unionList.removeAll(secondList);
   unionList.addAll(secondList);
   System.out.print("union = ");
   printList(unionList);
}
```

Правила за пренасяне и подравняване

Когато даден ред е дълъг, разделете го на два или повече реда, като редовете след първия отместете надясно с една табулация:

```
DictionaryEntry<K, V> newEntry =
   new DictionaryEntry<K, V>(key, value);
```

Грешно е да подравнявате сходни конструкции спрямо най-дългата от тях, тъй като това затруднява поддръжката на кода:

```
this.table      = new List[capacity];
this.loadFactor = loadFactor;
this.threshold  = (int) (this.table.length * this.loadFactor);
```

Грешно е да подравнявате параметрите при извикване на метод вдясно спрямо скобата за извикване:

```
System.out.printf("word '%s' is seen %d times in the text%n",
                  wordEntry.getKey(),
                  wordEntry.getValue());
```

Същият код може да се форматира правилно да се форматира по следния начин (този начин не е единственият правилен):

```
System.out.printf(
   "word '%s' is seen %d times in the text%n",
   wordEntry.getKey(),
   wordEntry.getValue());
```

Висококачествени методи

Качеството на нашите методи е от съществено значение за създаването на висококачествен софтуер и неговата поддръжка. Те правят програмите ни по-четливи и по-разбираеми. Методите ни помагат да намалим сложността на софтуера, да го направим по-гъвкав и по-лесен за модифициране.

От нас зависи, до каква степен ще се възползваме от тези предимства. Колкото по-високо е качеството на методите ни, толкова повече печелим от тяхната употреба. В следващите параграфи ще се запознаем с някои от основните принципи за създаване на качествени методи.

Защо да използваме методи?

Преди да започнем да говорим за добрите имена на методите, нека отделим известно време и да обобщим причините, поради които използваме методи.

Методът решава по-малък проблем. Много методи решават много малки проблеми. Събрани заедно, те решават по-голям проблем – това е римското правило "разделяй и владей" – по-малките проблеми се решават по-лесно.

Чрез методите се намалява сложността на задачата – сложните проблеми се разбиват на по-прости, добавя се допълнително ниво на абстракция, скриват се детайли за имплементацията и се намалява рискът от неуспех. С помощта на методите се избягва повторението на еднакъв код. Скриват се сложни последователности от действия.

Най-голямото предимство на методите е възможността за преизползване на код – те са най-малката преизползваема единица код. Всъщност точно така са възникнали методите.

Какво трябва да прави един метод?

Един метод трябва да върши работата, която е описана в името му и нищо повече. Ако един метод не върши това, което предполага името му, то или името му е грешно, или методът върши много неща едновременно, или просто методът е реализиран некоректно. И в трите случая методът не отговаря на изискванията за качествен програмен код и има нужда от преработка.

Един метод или трябва да свърши работата, която се очаква от него, или трябва да съобщи за грешка. В Java съобщаването за грешки се осъществява с хвърляне на изключение. При грешни входни данни е недопустимо даден метод да връща грешен резултат. Методът или трябва да работи

коректно или да съобщи, че не може да свърши работата си, защото не са на лице необходимите му условия (при некоректни параметри, неочаквано състояние на обектите и др.).

Например ако имаме метод, който прочита съдържанието на даден файл, той трябва да се казва **`readFileContents()`** и трябва да връща **`byte[]`** или **`String`** (в зависимост дали говорим за двоичен или текстов файл). Ако файлът не съществува или не може да бъде отворен по някаква причина, методът трябва да хвърли изключение, а не да върне празен низ или **`null`**. Връщането на неутрална стойност (например **`null`**) вместо съобщение за грешка не е препоръчителна практика, защото извикващият метод няма възможност да обработи грешката и изгубва носещото богата информация изключение.

Описаното правило има някои изключения. Обикновено то се прилага най-вече за публичните методи в класа. Те или трябва да работят коректно, или трябва да съобщят за грешка. При скритите (private) методи може да не се проверява за некоректни параметри, тъй като тези методи може да ги извика само авторът на класа, а той много добре знае какво подава като параметри и не винаги трябва да обработва изключителните ситуации, защото може да ги предвиди.

Един публичен метод или трябва да върши коректно точно това, което предполага името му, или трябва да съобщава за грешка. Всякакво друго поведение е некоректно.

Strong Cohesion и Loose Coupling

Правилата за логическа свързаност на отговорностите (strong cohesion) и за функционална независимост и минимална обвързаност с останалите методи и класове (loose coupling) важат с пълна сила за методите.

Вече обяснихме, че един метод трябва да решава един проблем, не няколко. Един метод не трябва да има странични ефекти или да решава няколко несвързани задачи, защото няма да можем да му дадем подходящо име, което пълно и точно го описва. Това означава, че всички методи, които пишем, трябва да имат strong cohesion, т.е. да са насочени към решаването на една единствена задача.

Методите трябва минимално да зависят от останалите методи и от класа, в който се намират и от останалите класове. Това свойство се нарича loose coupling.

В идеалния случай даден метод трябва да зависи единствено от параметрите си и да не използва никакви други данни като вход или като изход. Такива методи лесно могат да се извадят и да се преизползват в друг проект, защото са независими от средата, в която се изпълняват.

Понякога методите зависят от private променливи в класа, в който са дефинирани или променят състоянието на обекта, към който принадлежат. Това не е грешно и е нормално. В такъв случай говорим за обвързване (coupling) между метода и класа. Такова обвързване не е проблемно, защото целият клас може да се извади и премести в друг проект и ще започне да работи без проблем. Повечето класове от стандартните библиотеки на Java (от т. нар. Java API) дефинират методи, които зависят единствено от данните в класа, който ги дефинира и от подадените им параметри. В стандартните Java библиотеки зависимостите на методите от външни класове са минимални и затова тези библиотеки са лесни за използване.

Ако даден метод чете или променя глобални данни или зависи от още 10 обекта, които трябва да се инициализирани в инстанцията на неговия клас, той е силно обвързан с всички тези обекти. Това означава, че функционира сложно и се влияе от прекалено много външни условия и следователно възможността за грешки е голяма. Методи, които разчитат не прекалено много външни зависимости, са трудни за четене, за разбиране и за поддръжка. Силното функционално обвързване е лошо и трябва да се избягва, доколкото е възможно, защото води до код като спагети.

Колко дълъг да е един метод?

През годините са правени различни изследвания за оптималната дължина на методите, но в крайна сметка универсална формула за дължина на даден метод не съществува.

Практиката показва, че като цяло трябва да предпочитаме по-кратки методи (примерно не повече от един екран). Те са по-лесни за четене и разбиране, а вероятността да допуснем грешка при тях е значително по-малка.

Колкото по-голям е един метод, толкова по-сложен става той. Последващи модификации са значително по-трудни, отколкото при кратките методи и изискват много повече време. Тези фактори са предпоставка за допускане на грешки и по-трудна поддръжка.

Препоръчителната дължина на един метод е не-повече от един екран, но тази препоръка е само ориентировъчна. Ако методът се събира на екрана, той е по-лесен за четене, защото няма да се налага скролиране. Ако методът е по-дълъг от един екран, това трябва да ни накара да се замислим дали не можем да го разделим логически на няколко по-прости метода. Това не винаги е възможно да се направи по смислен начин, така че препоръката за дължината на методите е ориентировъчна.

Макар дългите методи да не са за предпочитане, това не трябва да е безусловна причина да разделяме на части даден метод само защото е дълъг. Методите трябва да са толкова дълги, колкото е необходимо.

Силната логическа свързаност на отговорностите при методите е много по-важна от дължината им.

Ако реализираме сложен алгоритъм и в последствие се получи дълъг метод, който все пак прави едно нещо и го прави добре, то в този случай дължината не е проблем.

Във всеки случай, винаги, когато даден метод стане прекалено дълъг, трябва да се замисляме, дали не е по-подходящо да изнесем част от кода в отделни методи, изпълняващи определени подзадачи.

Параметрите на методите

Едно от основните правила за подредба на параметрите на методите е основният или основните параметри да са първи. Пример:

```
public void archive(PersonData person, boolean persistent) {
```

Обратното би било доста по-объркващо:

```
public void archive(boolean persistent, PersonData person) {
```

Друго основно правило е имената на параметрите да са смислени. Честа грешка, особено при Java, е имената на параметрите да бъдат свързани с имената на типовете им. Пример:

```
public void archive(PersonData personData) {
```

Вместо нищо незначещото име **personData** (което носи информация единствено за типа), можем да използваме по-добро име (така е доста по-ясно кой точно обект архивираме):

```
public void archive(PersonData loggedUser) {
```

Ако има методи с подобни параметри, тяхната подредба трябва да е консистентна. Това би направило кода много по-лесен за четене:

```
public void archive(PersonData person, boolean persistent) {
   // ...
}

public void retrieve(PersonData person, boolean persistent) {
   // ...
}
```

Важно е да няма параметри, които не се използват. Те само могат да подведат ползвателя на този код.

Параметрите не трябва да се използват и като работни променливи – не трябва да модифицират. Ако модифицирате параметрите на методите, кодът става по-труден за четене и логиката му – по-трудна за проследяване. Винаги можете да дефинирате нова променлива вместо да променяте параметър. Пестенето на памет не е оправдание в този сценарий.

Неочевидните допускания трябва да се документират. Например мерната единица при подаване на числа. Например, ако имаме метод, който изчислява косинус от даден ъгъл, трябва да документираме дали ъгълът е в градуси или в радиани, ако това не е очевидно.

Броят на параметрите не трябва да надвишава 7. Това е специално, магическо число. Доказано е, че човешкото съзнание не може да следи повече от около 7 неща едновременно. Разбира се, тази препоръка е само за ориентир. Понякога се налага да предавате и много повече параметри. В такъв случай се замислете дали не е по-добре да ги предавате като някакъв клас с много полета. Например ако имате метода **`addStudent(…)`** с 15 параметъра (име, адрес, контакти и още много други), можете да намалите параметрите му като подавате групи логически свързани параметри като клас, примерно така: **`addStudent(personalData, contacts, universityDetails)`**. Всеки от новите 3 параметъра ще съдържа по няколко полета и пак ще се прехвърля същата информация, но в по-лесен за възприемане вид.

Понякога е логически по-издържано вместо един обект на метода да се подадат само едно или няколко негови полета. Това ще зависи най-вече от това дали методът трябва да знае за съществуването на този обект или не. Например имаме метод, който изчислява средния успех на даден студент – **`calcAverageResults(Student s)`**. Понеже успехът се изчислява от оценките на студента и останалите му данни нямат значение, е по-добре вместо Student да се предава като параметър списък от оценки. Така методът придобива вида **`calcAverageResults(List<Mark>)`**.

Правилно използване на променливите

В този параграф ще разгледаме няколко добри практики при локалната работа с променливи.

Връщане на резултат

Когато връщаме резултат от метод, той трябва да се запази в променлива преди да се върне. Следният пример не казва какво се връща като резултат:

```
return days * hoursPerDay * ratePerHour;
```

По-добре би било така:

```
int salary = days * hoursPerDay * ratePerHour;
return salary;
```

Има няколко причини да запазваме резултата преди да го видим. Едната е, че така документираме кода – по името на допълнителната променлива става ясно какво точно връщаме. Другата причина е, че когато дебъгваме програмата, ще можем да я спрем в момента, в който е изчислена връщаната стойност и ще можем да проверим дали е коректна. Третата причина е, че избягваме сложните изрази, които понякога може да са няколко реда дълги и заплетени.

Принципи при инициализиране

В Java всички член-променливи в класовете се инициализират автоматично още при деклариране (за разлика от C/C++). Това се извършва от виртуалната машина. Така се избягват грешки с неправилно инициализирана памет. Всички променливи, сочещи обекти (reference type variable) се инициализират с **null**, а всички примитивни типове – с 0 (**false** за **boolean**).

Компилаторът задължава всички локални променливи в кода на една програма да бъдат инициализирани изрично преди употреба, иначе връща грешка при компилация. Ето един пример, който ще предизвика грешка при компилация, защото се прави опит за използване на неинициализирана променлива:

```
int value;
System.out.println(value);
```

При опит за компилация се връща грешка на втория ред:

```
/>javac Program.java
Program.java:11: variable value might not have been initialized
                System.out.println(value);
                                   ^
1 error
```

Ето още един малко по-сложен пример:

```
int value;
if(<condition1>) {
   if(<condition2>) {
      value = 1;
   }
} else {
   value = 2;
}
System.out.println(value);
```

За щастие компилаторът е достатъчно интелигентен и хваща подобни "недоразумения" – отново същата грешка.

Забележете следната особеност: ако сложим **`else`** на вложения **`if`** в горния код, всичко ще се компилира. Компилаторът проверява всички възможни пътища, по които може да мине изпълнението и ако при всеки един от тях има инициализация на променливата, той не връща грешка и променливата се инициализира правилно.

Добрата практика е всички променливи да се инициализират изрично още при деклариране:

```
int value = 0;
Student intern = null;
```

Инициализиране на член-променливи на обекти

Някои обекти, за да бъдат правилно инициализирани, трябва да имат стойности на поне няколко техни полета. Например обект от тип Човек, трябва да има стойност на полетата "име" и "фамилия". Това е проблем, от който компилаторът не може да ни опази.

Единият начин да бъде решен този проблем е да се премахне конструкторът по подразбиране (конструкторът без параметри) и на негово място да се сложат един или няколко конструктора, които получават достатъчно данни (във формата на параметри) за правилното инициализиране на съответния обект.

Това решение, колкото и да е добро, противоречи на една от основните директиви на JavaBeans спецификацията, а именно всеки обект да има конструктор по подразбиране. Причината за подобно изискване е, че JavaBeans обектите са направени така, че да могат да се използват автоматично от различни библиотеки. Липсата на конструктор по подразбиране би попречила на тези инструменти да създават бързо и лесно такива обекти.

Тази директива не се спазва винаги, но става по-популярна заради все повече нови инструменти за автоматична работа с обекти. Например последната библиотека в Java 6 за работа с уеб услуги – JAX-WS – има подобно изискване.

Решение би било всеки, които ползва такива обекти, да проверява за пълнотата на тези обекти (погледнете секцията за защитно програмиране) или обектите да предоставят метод, който проверява това.

Деклариране на променлива в блок/метод

Съгласно конвенцията за писане на код на Sun, една променлива трябва да се декларира в началото на блока или тялото на метода, в който се намира:

```
static int archive() {
   int result = 0;        // beginning of method body

   if (<condition>) {
      int result = 0;      // beginning of "if" block
      // .. Code ...
   }
}
```

Изключение правят променливите, които се декларират в инициализиращата част на **for** цикъла:

```
for (int i = 0; i < data.length; i++) { ...
```

Повечето добри програмисти предпочитат да декларират една променлива максимално близо до мястото, на което тя ще бъде използвана и по този начин да намалят нейния живот (погледнете следващия параграф) и същевременно възможността за грешка. Този подход е препоръчителен пред конвенцията на Sun.

Обхват, живот, активност

Понятието **обхват на променлива** (**variable scope**) всъщност описва колко "известна" е една променлива. В Java тя може да бъде (подредени в низходящ ред) статична променлива, член-променлива (на клас) и локална променлива (в метод).

Колкото по-голям е обхватът на дадена променлива, толкова по-голяма е възможността някой да се обвърже с нея и така да увеличи своя coupling, което не е хубаво. Следователно обхватът на променливите трябва да е възможно най-малък.

Добър подход при работата с променливи е първоначално те да са с минимален обхват. При необходимост той да се разширява. Така по естествен начин всяка променлива получава необходимия за работата ѝ обхват. Ако не знаете какъв обхват да ползвате, започвайте от **private** и при нужда преминавайте към **protected** или **public**.

Статичните променливи е най-добре да са винаги **private** и достъпът до тях да става контролирано, чрез извикване на подходящи методи.

Ето един пример за лошо семантично обвързване със статична променлива – ужасно лоша практика:

```
public class Globals {
   public static int state = 0;
}
```

```
public class Genious {
   public static void printSomething() {
      if (Globals.state == 0) {
         System.out.println("Hello.");
      } else {
         System.out.println("Good bye.");
      }
   }
}
```

Ако променливата **state** беше дефинирана като **private**, такова обвързване нямаше да може да се направи, поне не директно.

Диапазон на активност (**span**) е средният брой линии между обръщенията към дадена променлива. Той зависи от гъстотата на редовете код, в които тази променлива се използва. Диапазонът на променливите трябва да е минимален. По тази причина променливите трябва да се декларират и инициализират възможно най-близко до мястото на първата им употреба, а не в началото на даден метод или блок. Ето един пример за неправилно използване на променливи (излишно голям диапазон на активност):

```
int count;
int[] numbers = new int[100];
for (int i=0; i<numbers.length; i++) {
   numbers[i] = i;
}
count = 0;
for (int i=0; i<numbers.length/2; i++) {
   numbers[i] = numbers[i] * numbers[i];
}

for (int i=0; i<numbers.length; i++) {
   if (numbers[i] % 3 == 0) {
      count++;
   }
}
System.out.println(count);
```

В този пример променливата **count** служи за преброяване на числата, които се делят без остатък на 3 и се използва само в последния **for** цикъл. Тя е дефинирана излишно рано и се инициализира много преди да има нужда от инициализацията. Ако трябва да се преработи този код, за да се намали диапазонът на активност на променливата **count**, той ще добие следния вид:

```
int[] numbers = new int[100];
for (int i=0; i<numbers.length; i++) {
   numbers[i] = i;
```

```
}
for (int i=0; i<numbers.length/2; i++) {
   numbers[i] = numbers[i] * numbers[i];
}

int count = 0;
for (int i=0; i<numbers.length; i++) {
   if (numbers[i] % 3 == 0) {
      count++;
   }
}
System.out.println(count);
```

Живот (**lifetime**) на една променлива е обемът на кода от първото до последното й реферирane в даден метод. В тази дефиниция имаме предвид само локални променливи, понеже член-променливите живеят докато съществува класът, в който са дефинирани, а статичните променливи – докато съществува виртуалната машина.

Важно е програмистът да следи къде се използва дадена променлива, нейният диапазон на активност и период на живот. Основното правило е да се направят обхватът, животът и активността на променливите колкото се може по-малки. От това следва едно важно правило:

Декларирайте локалните променливи възможно най-късно, непосредствено преди да ги използвате за първи път, и ги инициализирайте заедно с декларацията им.

Променливите с по-голям обхват и по-дълъг живот, трябва да имат по-описателни имена, примерно **totalStudentsCount**. Причината е, че те ще бъдат използвани на повече места и за по-дълго време и за какво служат няма да бъде ясно от контекста. Променливите с живот няколко реда могат да бъдат с кратко и просто име, примерно **count**. Те нямат нужда от дълги и описателни имена, защото техният смисъл е ясен от контекста, в който се използват, а този контекст е твърде малък (няколко реда), за да има двусмислия.

Работа с променливи – още правила

Една променлива трябва да се използва само за една цел. Това е много важно правило. Извиненията, че ако се преизползва едно променлива за няколко цели се пести на памет, в общия случай не са добро оправдание. Ако една променлива се ползва за няколко съвсем различни цели, какво име ще й дадем? Например, ако една променлива се използва да брои студенти и в някои случаи техните оценки, то как ще я кръстим: **count**, **studentsCount**, **marksCount** или **StudentsOrMarksCount**?

Ползвайте една променлива само за една единствена цел. Иначе няма да можете да й дадете подходящо име.

Никога не трябва да има променливи, които не се използват. В такъв случай тяхното дефиниране е било безсмислено. За щастие сериозните среди за разработка на Java приложения (например Eclipse, но и не само) издават предупреждение за подобни "нередности".

Трябва да се избягват и променливи със скрито значение. Например Пешо е оставил променливата **X**, за да бъде видяна от Митко, който трябва да се сети да имплементира още един метод, в който ще я ползва.

Правилно използване на изрази

При работата с изрази има едно много просто правило: не ползвайте сложни изрази! Сложен израз наричаме всеки израз, който извършва повече от едно действие. Ето пример за сложен израз:

```
for (int i=0; i<xCoord.length; i++) {
   for (int j=0; j<yCoord.length; j++) {
      matrix[i][j] =
         matrix[xCoord[findMax(i)+1]][yCoord[findMin(i)+1]] *
         matrix[yCoord[findMax(i)+1]][xCoord[findMin(i)+1]];
   }
}
```

В примерния код имаме сложно изчисление, което запълва дадена матрица спрямо някакви изчисления върху някакви координати. Всъщност е много трудно да се каже какво точно се случва, защото е използван сложен израз.

Има много причини, заради които трябва да избягваме използването на сложни изрази като в примера по-горе. Ще изброим някои от тях:

- Кодът трудно се чете. В нашия пример няма да ни е лесно да разберем какво прави този код и дали е коректен.
- Кодът трудно се поддържа. Помислете, какво ще ни струва да поправим грешка в този код, ако не работи коректно.
- Кодът трудно се поправя, ако има дефекти. Ако примерният код по-горе даде **ArrayIndexOutOfBoundsException**, как ще разберем извън границите на кой точно масив сме излезли? Това може да е масивът **xCoord** или **yCoord** или **matrix**, а излизането извън тези масиви може да е на няколко места.
- Кодът трудно се дебъгва. Ако намерим грешка, как ще дебъгнем изпълнението на този израз, за да намерим грешката?

Всички тези причини ни подсказват, че писането на сложни изрази е вредно и трябва да се избягва. Вместо един сложен израз можем да напишем

няколко по-прости изрази и да ги запишем в променливи с разумни имена. По този начин кодът става по-прост, по-ясен, по-лесен за четене и разбиране, по-лесен за промяна, по-лесен за дебъгване и по-лесен за поправяне. Нека сега пренапишем горния код, без да използваме сложни изрази:

```
for (int i=0; i<xCoord.length; i++) {
   for (int j=0; j<yCoord.length; j++) {
      int maxStartIndex = findMax(i) + 1;
      int minStartIndex = findMax(i) - 1;
      int minXcoord = xCoord[minStartIndex];
      int maxXcoord = xCoord[maxStartIndex];
      int minYcoord = yCoord[minStartIndex];
      int maxYcoord = yCoord[maxStartIndex];
      matrix[i][j] =
         matrix[maxXcoord][minYcoord] *
         matrix[maxYcoord][minXcoord];
   }
}
```

Забележете колко по-прост и ясен стана кода. Наистина, без да знаем какво точно изчисление извършва този код, ще ни е трудно да го разберем, но ако настъпи изключение, лесно ще намерим на кой ред възниква и чрез дебъгера можем да проследим защо се получава и евентуално да го поправим.

Не пишете сложни изрази. На един ред трябва да се извършва по една операция. Иначе кодът става труден за четене, за поддръжка, за дебъгване и за промяна.

Използване на константи

В добре написания програмен код не трябва да има "магически числа" и стрингове. Такива наричаме всички литерали в програмата, които имат стойност, различно от **0**, **1**, **-1**, **""** и **null** (с дребни изключения).

За да обясним по-добре концепцията за използване на именувани константи, ще дадем един пример за код, който има нужда от преработка:

```
public class MathUtils {

   public static double calcCircleArea(double radius) {
      double area = 3.14159206 * radius * radius;
      return area;
   }

   public static double calcCirclePerimeter(double radius) {
```

```
      double perimeter = 2 * 3.14159206 * radius;
      return perimeter;
   }

   public static double calcElipseArea(double axis1,
         double axis2) {
      double area = 3.14159206 * axis1 * axis2;
      return area;
   }

}
```

В примера използваме три пъти числото **3.14159206** (Π), което е повторение на код. Ако решим да променим това число, като го запишем например с по-голяма точност, ще трябва да променим програмата на три места. Възниква идеята да дефинираме това число като стойност, която е глобална за програмата и не може да се променя. Именно такива стойности в Java се декларират като именувани константи по следния начин:

```
public static final double PI = 3.14159206;
```

След тази декларация константата **PI** е достъпна от цялата програма и може да се ползва многократно. При нужда от промяна променяме само на едно място и промените се отразяват навсякъде. Ето как изглежда нашия примерен клас **MathUtils** след изнасянето на числото **3.14159206** в константа:

```
public class MathUtils {
   public static final double PI = 3.14159206;

   public static double calcCircleArea(double radius) {
      double area = PI * radius * radius;
      return area;
   }

   public static double calcCirclePerimeter(double radius) {
      double perimeter = 2 * PI * radius;
      return perimeter;
   }

   public static double calcElipseArea(double axis1,
         double axis2) {
      double area = PI * axis1 * axis2;
      return area;
   }
}
```

Кога да използваме константи?

Използването на константи помага да избегнем използването на "магически числа" и стрингове в нашите програми и позволява да дадем имена на числата и стринговете, които ползваме. В предходния пример не само избегнахме повторението на код, но и документирахме факта, че числото **`3.14159206`** е всъщност добре известната в математиката константа **`PI`**.

Константи трябва да дефинираме винаги, когато имаме нужда да ползваме числа или символни низове, за които не е очевидно от къде идват и какъв е логическият им смисъл. Константи е нормално да дефинираме и за всяко число или символен низ, който се ползва повече от веднъж в програмата.

Ето няколко типични ситуации, в които трябва да ползвате именувани константи:

- За имена на файлове, с които програмата оперира. Те често трябва да се променят и затова е много удобно да са изнесени като константи в началото на програмата.
- За константи, участващи в математически формули и преобразувания. Доброто име на константата подобрява шансът при четене на кода да разберете смисъла на формулата.
- За размери на буфери или блокове памет. Тези размери може да се наложи да се променят и е удобно да са изнесени като константи. Освен това използването на константата **`READ_BUFFER_SIZE`** вместо някакво магическо число 8192 прави кода много по-ясен и разбираем.

Кога да не използваме константи?

Въпреки, че много книги препоръчват всички числа и символни низове, които не са **`0`**, **`1`**, **`-1`**, **`""`** и **`null`** да бъдат изнасяни като константи, има някои изключения, в които изнасянето на константи е вредно. Запомнете, че изнасянето на константи се прави, за да се подобри четимостта на кода и поддръжката му във времето. Ако изнасянето на дадена константа не подобрява четимостта на кода, няма нужда да го правите.

Ето някои ситуации, в които изнасянето на текст или магическо число като константа е вредно:

- Съобщения за грешки и други съобщения към потребителя (примерно "въведете името си"): изнасянето им затруднява четенето на кода вместо да го улесни.
- SQL заявки (ако използвате бази от данни, командите за извличане на информацията от базата данни се пише на езика SQL и представлява стринг). Изнасянето на SQL заявки като константи прави четенето на кода по-трудно и не се препоръчва.

- Заглавия на бутони, диалози, менюта и други компоненти от потребителския интерфейс също не се препоръчва да се изнасят като константи, тъй като това прави кода по-труден за четене.

В Java съществуват библиотеки, които подпомагат интернационализацията и позволяват да изнасяте съобщения за грешки, съобщения към потребителя и текстовете в потребителския интерфейс в специални ресурсни файлове, но това не са константи. Такъв подход се препоръчва, ако програмата, която пишете ще трябва да се интернационализира.

Използвайте именувани константи, за да избегнете използването и повтарянето на магически числа и стрингове в кода и най-вече, за да подобрите неговата четимост. Ако въвеждането на именувана константа затруднява четимостта на програмата, по-добре оставете твърдо зададената стойност в кода!

Правилно използване на конструкциите за управление

Конструкциите за управление са циклите и условните конструкции. Сега ще разгледаме добрите практики за правилното им използване.

Правилно използване на условни конструкции

Условни конструкции в Java са **`if-else`** операторите и **`switch-case`** операторите.

При **`if-else`** винаги е било добра практика тялото на тази условна конструкция да се огражда с къдрави скоби:

```
if (condition) {

} else {

}
```

Скобите, разбира се, могат да се пропуснат, ако има само по един оператор в тялото на конструкцията, но това е опасно, защото при грешно форматиране и добавяне на допълнителен оператор на пръв поглед може да изглежда, че и двата са в условната конструкция, но да не са:

```
if (condition)
   doSomething();
   doSomethingElse();
doDifferentThing();
```

В този пример вторият метод не е в условната конструкция, но на пръв поглед изглежда, че е в нея.

При **switch-case** конструкциите винаги се препоръчва да се използва **break** след края на всяка **case** конструкция. Липсата на **break** може да доведе до много грешки.

Какъв би бил резултатът на следния пример?

```
int value = 1;
switch (value) {
case 1:
   System.out.println("One");
case 2:
   System.out.println("Two");
case 3:
   System.out.println("Three");
default:
   System.out.println("default");
}
```

Резултатът малко неочаквано е:

```
One
Two
Three
default
```

Причината е, че накрая на всеки **case** няма **break**. Кодът трябва да изглежда така:

```
int value = 1;
switch (value) {
case 1:
   System.out.println("One");
   break;
case 2:
   System.out.println("Two");
   break;
case 3:
   System.out.println("Three");
   break;
default:
   System.out.println("default");
   break;
}
```

Препоръчва се **default** секцията винаги да е последна, най-отдолу.

Дълбокото влагане на if-конструкции е лоша практика, защото прави кода сложен и труден за четене. Ето един пример:

```
if (maxElem != Integer.MAX_VALUE) {
   if (arr[i] < arr[i + 1]) {
      if (arr[i + 1] < arr[i + 2]) {
         if (arr[i + 2] < arr[i + 3]) {
            maxElem = arr[i + 3];
         } else {
            maxElem = arr[i + 2];
         }
      } else {
         if (arr[i + 1] < arr[i + 3]) {
            maxElem = arr[i + 3];
         } else {
            maxElem = arr[i + 1];
         }
      }
   } else {
      if (arr[i] < arr[i + 2]) {
         if (arr[i + 2] < arr[i + 3]) {
            maxElem = arr[i + 3];
         } else {
            maxElem = arr[i + 2];
         }
      } else {
         if (arr[i] < arr[i + 3]) {
            maxElem = arr[i + 3];
         } else {
            maxElem = arr[i];
         }
      }
   }
}
```

Този код е напълно нечетим. Причината е, че има прекалено дълбоко влагане на **if** конструкциите една в друга. За да се подобри четимостта на този код, може да се въведат един или няколко метода, в които да се изнесе част от сложната логика. Ето как може да се преработи кода, за да се намали вложеността на условните конструкции и да стане по-разбираем:

```
if (maxElem != Integer.MAX_VALUE) {
   maxElem = findMax(arr, i);
}

private static int findMax(int[] arr, int i) {
   if (arr[i] < arr[i + 1]) {
      int maxElem = max(arr[i + 1], arr[i + 2], arr[i + 3]);
```

```
      return maxElem;
   } else {
      int maxElem = max(arr[i], arr[i + 2], arr[i + 3]);
      return maxElem;
   }
}

private static int max(int i, int j, int k) {
   if (i < j) {
      int maxElem = max(j, k);
      return maxElem;
   } else {
      int maxElem = max(i, k);
      return maxElem;
   }
}

private static int max(int i, int j) {
   if (i < j) {
      return j;
   } else {
      return i;
   }
}
```

Изнасянето на част от кода в отделен метод и най-лесния и ефективен начин да се намали вложеността на група условни конструкции, като се запази логическият им смисъл.

Правилно използване на цикли

Правилното използване на различните конструкции за цикли е от значение при създаването на качествен софтуер. В следващите параграфи ще се запознаем с някои принципи, които ни помагат да определим кога и как да използваме определен вид цикъл.

Избиране на подходящ вид цикъл

Ако в дадена ситуация не можем да решим дали да използваме **`for`**, **`while`** или **`do-while`** цикъл, можем лесно да решим проблема, придържайки се към следващите принципи:

Ако се нуждаем от цикъл, който да се изпълни определен брой пъти, то е добре да използваме **`for`** цикъл. Този цикъл се използва в прости случаи, когато не се налага да прекъсваме изпълнението. При него още в началото задаваме параметрите на цикъла и в общия случай, в тялото не се грижим за контрола му. Стойността на брояча вътре в тялото на цикъла не трябва да се променя.

Ако е необходимо да следим някакви условия, при които да прекратим изпълнението на цикъла, тогава вероятно е по-добре да използваме **while** цикъл. **while** цикълът е подходящ в случаи, когато не знаем колко точно пъти трябва да се изпълни тялото цикъла. При него изпълнението продължава, докато не се достигне дадено условие за край. Ако имаме налице предпоставките за използване на **while** цикъл, но искаме да сме сигурни, че тялото ще се изпълни поне веднъж, то в такъв случай трябва да използваме **do-while** цикъл.

Не влагайте много цикли

Както и при условните конструкции, и при циклите е лоша практика да имаме дълбоко влагане. Дълбокото влагане обикновено се получава от голям брой цикли и условни конструкции, поставени една в друга. Това прави кода сложен и труден за четене и поддръжка. Такъв код лесно може да се подобри, като се отдели част от логиката в отделен метод. Съвременните среди за разработка могат да правят такава преработка на кода автоматично (ще обясним за това в секцията за преработка на кода).

Защитно програмиране

Защитно програмиране (defensive programming) е термин обозначаващ практика, която е насочена към защита на кода от некоректни данни. Защитното програмиране пази кода от грешки, които никой не очаква. То се имплементира чрез проверка на коректността на всички входни данни. Това са данните, идващи от външни източници, входните параметри на методите, конфигурационни файлове и настройки, данни въведени от потребителя, дори и данни от друг локален метод.

Защитното програмиране изисква всички данни да се проверяват, дори да идват от източник, на когото се вярва. По този начин, ако в този източник има грешка (бъг), то тя ще бъде открита по-бързо.

Защитното програмиране се имплементира чрез assertions, изключения и други средства за управление на грешки.

Assertions

Това е специална конструкция в Java, която позволява имплементацията на защитно програмиране. Появяват се в JDK 1.4. Ето един бърз пример:

```
public int archive(PersonData user, boolean persistent) {
   assert user != null;

   // Do some processing
   int resultFromProcessing = ...

   assert resultFromProcessing >= 0 :
```

```
      "resultFromProcessing is negative. There is a bug";

   return resultFromProcessing;
}
```

От кода се виждат два различни начина на употреба на **assert**:

```
assert <condition>;
```

и

```
assert <condition> : <message>;
```

Във втория вариант има допълнителен обяснителен текст.

Основната идея на тази конструкция е да достави код, който е по-четим и от който бъговете се изчистват по-бързо по време на разработка. Конструкцията се слага на места, на които имаме някакви ограничителни условия за дадена променлива. В нашия пример, ако методът **archive** се използва само вътрешно, то не би трябвало никой да го извика с **null** вместо с инстанция на **PersonData**.

Слагайки **assert** все едно казваме "тук със сигурност тази променлива не е **null**". Ако някой **assert** не мине (т.е. условието в него не е изпълнено), се генерира грешка от тип **AssertionException**.

Assertions могат да се изключват. По замисъл те трябва да са включени само по време на разработка, докато се открият всички бъгове. Когато бъдат изключени всички проверки в тях спират да се изпълняват. Идеята на изключването е, че след края на разработката, тези проверки не са повече нужни и само забавят софтуера.

Assertions по подразбиране са изключени. За да се включат, се подава специален параметър на виртуалната машина **-ea**:

```
java -ea introjavabook.Storage
```

Ако дадена проверка е смислено да продължи да съществува след края на разработката (примерно проверява входни данни на метод, които идват от потребителя), то тази проверка е неправилно имплементирана с assertions и трябва да бъде имплементирана с изключения.

Assertions се използват само на места, на които трябва дадено условие да бъде изпълнено и единствената причина да не е, е да има бъг в програмата.

Защитно програмиране с изключения

Изключенията (exceptions) предоставят мощен механизъм за централизирано управление на грешки и непредвидени ситуации. В главата "Обработка на изключения" те са описани подробно.

Изключенията позволяват проблемните ситуации да се обработват на много нива. Те улесняват писането и поддръжката на надежден програмен код.

Разликата между изключенията и assertions е в това, че изключенията в защитното програмиране се използват най-вече за защитаване на публичния интерфейс на един компонент. Този механизъм се нарича **fail-safe** (в свободен превод "проваляй се грациозно" или "подготвен за грешки").

Ако методът **archive**, описан малко по-нагоре, беше част от публичния интерфейс на архивиращ компонент, а не вътрешен метод, то този метод би трябвало да бъде имплементиран така:

```
public int archive(PersonData user, boolean persistent) {
   if (user == null)
      throw new StorageException("null parameter");

   // Do some processing
   int resultFromProcessing = ...

   assert resultFromProcessing >= 0 :
      "resultFromProcessing is negative. There is a bug";

   return resultFromProcessing;
}
```

Вторият assert остава, тъй като той е предвиден за променлива създадена вътре в метода.

Изключенията трябва да се използват, за да се уведомят другите части на кода за проблеми, които не трябва да бъдат игнорирани. Хвърлянето на изключение е оправдано само в ситуации, които наистина са изключителни и трябва да се обработят по някакъв начин. За повече информация за това кои ситуации са изключителни и кои не погледнете главата "Обработка на изключения".

Ако даден проблем може да се обработи локално, то обработката трябва да се направи в самия метод и изключение не трябва да се хвърля. Ако даден проблем не може да се обработи локално, той трябва да бъде прехвърлен към извикващия метод чрез **throws** декларация.

Трябва да се хвърлят изключения с подходящо ниво на абстракция. Пример: **getEmployeeInfo()** може да хвърля **EmployeeException**, но не и **FileNotFoundException**. Погледнете последният пример, той хвърля **StorageException**, а не **NullPointerException**.

Повече за добрите практики при управление на изключенията можете да прочетете от секцията "Добри практики при работа с изключения" на главата "Обработка на изключения".

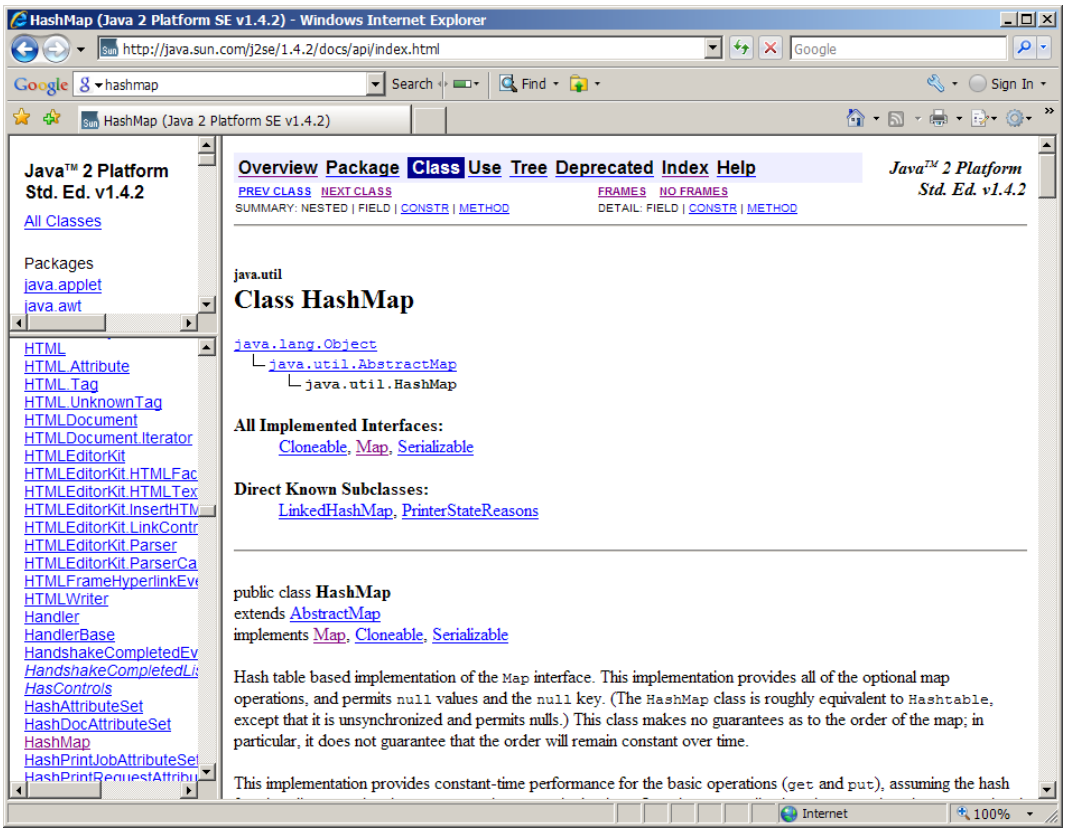
Документация на кода

В Java има специална нотация за писане на коментари. Нарича се JavaDoc. Ето един пример:

```
/**
 * A class representing a thread pool. It works with {@link
 * Thread}s that it keeps alive for reuse.
 *
 * Usage:
 * <code>
 *   TreadPool pool = new ThreadPool();
 * </code>
 *
 * @author Mihail Stoynov
 * @version 1.0
 * @see java.lang.Thread
 */
public class ThreadPool {
   /** Some comment here */
   public static final int MAX_POOL_SIZE = 16;
}
```

JavaDoc коментарите се различават от обикновените коментари. Те започват с **/**** вместо с **/***. JavaDoc коментарите няма съкратен запис за разлика от обикновените коментари (**//Text**). Забележете възможността да пишете HTML тагове директно в документацията – той ще се появи на генерираните страници. JavaDoc позволява и група специални думи, които започват с **@**. За тях можете да научите повече от документацията на Java на сайта на Sun.

От JavaDoc документацията могат автоматично да се създават HTML страници със съдържанието на коментарите, което е много полезно:



Самодокументиращ се код

Коментарите в кода не са основният източник на документация. Запомнете това! Добрият стил на програмиране е най-добрата документация! Самодокументиращ се код е такъв, на който лесно се разбира основната му цел, без да е необходимо да има коментари.

Най-добрата документация на кода е да пишем качествен код. Лошият код не трябва да се коментира, а трябва да се пренапише, така че сам да описва себе си. Коментарите в програмата само допълват документацията на добре написания код.

Характеристики на самодокументиращия се код

Характеристики на самодокументиращия се код са добра структура на програмата – подравняване, организация на кода, използване на ясни и лесни за разбиране конструкции, избягване на сложни изрази. Такива са още употребата на подходящи имена на променливи, методи и класове и употребата на именувани константи, вместо "магически" константи и

текстови полета. Реализацията трябва да е опростена максимално, така че всеки да я разбере.

Самодокументиращ се код – важни въпроси

Въпроси, които трябва да си зададем преди да отговорим на въпроса дали кодът е самодокументиращ се:

- Подходящо ли е името на класа и показва ли основната му цел?
- Става ли ясно от интерфейса как трябва да се използва класа?
- Показва ли името на метода основната му цел?
- Всеки метод реализира ли една добре определена задача?
- Имената на променливите съответстват ли на тяхната употреба?
- Групирани ли са свързаните един с друг оператори?
- Само една задача ли изпълняват конструкциите за итерация (циклите)?
- Има ли дълбоко влагане на условни конструкции?
- Показва ли организацията на кода неговата логическата структура?
- Дизайнът недвусмислен и ясен ли е?
- Скрити ли са детайлите на имплементацията възможно най-много?

"Ефективни" коментари

Коментарите понякога могат да навредят повече, отколкото да помогнат. Добрите коментари не повтарят кода и не го обясняват – те изясняват неговата идея. Коментарите трябва да обясняват на по-високо ниво какво се опитваме да постигнем. Писането на коментари помага да осмислим по-добре това, което искаме да реализираме.

Ето един пример за лоши коментари, които повтарят кода и вместо да го направят по-лесно четим, го правят по-тежък за възприемане:

```
public static ArrayList<Integer> getPrimes(int start, int end) {
   // Create new list of integers
   ArrayList<Integer> primesList = new ArrayList<Integer>();

   // Perform a loop from start to end
   for (int num = start; num <= end; num++) {

      // Declare boolean variable, initially true
      boolean prime = true;

      // Perform loop from 2 to sqrt(num)
      for (int div = 2; div <= Math.sqrt(num); div++) {
```

```
        // Check if div divides num with no remainder
        if (num % div == 0) {

           // We found a divider -> the number is not prime
           prime = false;

           // Exit from the loop
           break;
        }

        // Continue with the next loop value
     }

     // Check if the number is prime
     if (prime) {

        // Add the number to the list of primes
        primesList.add(num);
     }
   }

   // return the list of primes
   return primesList;
}
```

Ако вместо да слагаме наивни коментари, ги ползваме, за да изясним неочевидните неща в кода, те могат да са много полезни. Вижте как бихме могли да коментираме същия код, така че да му подобрим четимостта:

```
/**
 * @return a list of all primes in given range [start, end].
 * A number num is prime if it can not be divided to any number
 * in the range [2, sqrt(num)]. We check condition this for all
 * numbers in the given range.
 */
public static ArrayList<Integer> getPrimes(int start, int end) {
   ArrayList<Integer> primesList = new ArrayList<Integer>();
   for (int num = start; num <= end; num++) {
      boolean prime = true;
      for (int div = 2; div <= Math.sqrt(num); div++) {
         if (num % div == 0) {
            // Found a divider -> num is not prime
            prime = false;
            break;
         }
      }
```

```
      if (prime) {
         primesList.add(num);
      }
   }
   return primesList;
}
```

В случая единственият неочевиден въпрос е защо пробваме да търсим делители в диапазона от 2 до **sqrt(num)**. Ако се слага коментар, той трябва да изясни този въпрос. Останалото е очевидно от кода. Имената на променливите са ясни и сами говорят за себе си. Логиката на кода е очевидна и няма нужда от коментари. Достатъчно е да се опише за какво служи даденият метод и основната му идея (как работи) в едно изречение.

При писането на "ефективни" коментари е добра практика да се използва псевдокод, когато е възможно. Коментарите трябва да се пишат, когато се създава самия код, а не след това.

Продуктивността никога не е добра причина, за да не се пишат коментари. Трябва да се документира всичко, което не става ясно от кода. Поставянето на излишно много коментари е толкова вредно колкото и липсата на такива.

Лошият код не става по-добър с повече коментари. За да стане добър код, просто трябва да се преработи.

Преработка на кода (Refactoring)

Терминът Refactoring се появява през 1993 и е популяризиран от Мартин Фаулър в едноименната му книга по темата. В тази книга се разглеждат много техники за преработка на код. Нека и ние разгледаме няколко.

Дадена програма се нуждае от преработка, при повторение на код. Повторението на код е опасно, защото когато трябва да се променя, трябва да се променя на няколко места и естествено някое от тях може да бъде пропуснато и така да се получи несъответствие. Избягването на повтарящ се код може да стане чрез изваждане на метод или преместване на код от клас-наследник в базов клас.

Преработка се налага и при методи, които са нараснали с времето. Прекалената дължината на метод е добра причина да се замислим дали методът не може да се раздели логически на няколко по-малки и по-прости метода.

При цикъл с прекалено дълбоко ниво на влагане трябва да се замислим дали не можем да извадим в отделен метод част от кода му. Обикновено това подобрява четимостта на кода и го прави по-лесен за разбиране.

Преработката е наложителна при клас, който изпълнява несвързани отговорности (poor cohesion). Клас, който не предоставя достатъчно добро ниво на абстракция също трябва да се преработи.

Дългият списък с параметри и публичните полета също трябва да са в графата "да се поправи". Тази графа трябва да допълни и когато една промяна налага да се променят паралелно още няколко класа. Прекалено свързани класове или недостатъчно свързани класове също трябва да се преработят.

Преработка на код на ниво данни

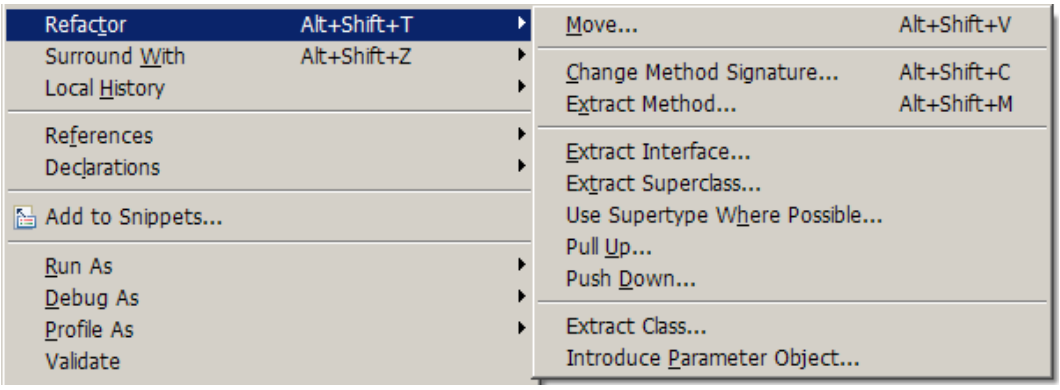
Добра практика е в кода да няма "магически" числа. Те трябва да бъдат заменени с константи. Променливите с неясни имена трябва да се преименуват. Дългите условни изрази могат да бъдат преработени в отделни методи. За резултата от сложни изрази могат да се използват междинни променливи. Група данни, които се появяват заедно могат да се преработят в отделен клас. Свързаните константи е добре да се преместят в изброими типове (enumerations).

Добра практика е всички задачи от един по-голям метод, които не са свързани с основната му цел, да се "преместят" в отделни методи (extract method). Сходни задачи трябва да се групират в общи класове, сходните класове – в общ пакет. Ако група класове имат обща функционалност, то тя може да се изнесе в базов клас.

Не трябва да има циклични зависимости между класовете – те трябва да се премахват. Най-често по-общият клас има референция към по-специализирания (връзка родител-деца).

Refactoring с Eclipse

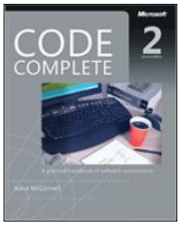
Средата Eclipse предоставя едни от най-мощните инструменти за преработка на код. Почти всички описани досега операции могат да се извършват автоматично, а това ще намали осезаемо риска от грешки. Менютата Refactoring и Source дават предостатъчно възможности за поправка във всеки един клас:



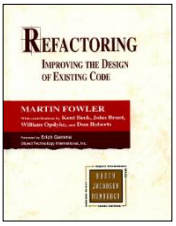
Преименуването на променливи, автоматичното създаване на методи за достъп (getters and setters), автоматичното обособяване на функционалност от един метод в друг, изваждането на променливи в константи са само малка част от възможностите на Eclipse.

Препоръчително е дори за прости операции като преименуване на променлива да се използва инфраструктурата на средата за програмиране, а не да се прави ръчно, тъй като тя се грижи за всички използвания на тази променлива дори извън класа и така се избягва рискът от грешки.

Ресурси



Библията за качествен програмен код се казва "Code Complete" и през 2004 година излезе във второ издание. Авторът й Стийв Макконъл е световноизвестен експерт по писане на качествен софтуер. В книгата можете да откриете много повече примери и детайлни описания на различни проблеми, които не успяхме да разгледаме.



Друга добра книга е "Refactoring" на Мартин Фаулър. Тази книга се смята за библията в преработката на код. В нея за първи път са описани понятията "extract method" и други, стоящи в основата на съвременните шаблони за преработка на съществуващ код.

Упражнения

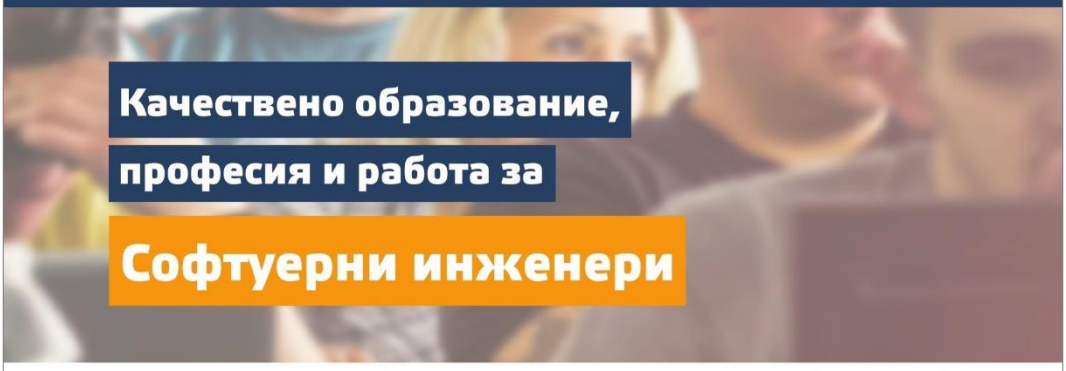
1. Вземете кода от първия пример в тази глава и го направете качествен.
2. Прегледайте собствения си код досега и вижте какви грешки допускате. Обърнете особено внимание на тях и помислете защо ги допускате.
3. Отворете чужд код и се опитайте само на базата на кода и документацията да разберете какво прави той. Има ли неща, които не ви стават ясни от първия път? А от втория?
4. Разгледайте класове от Java API-то. Намирате ли примери за некачествен код?
5. Ползвали ли сте (виждали ли сте) някакви код конвенции. През призмата на тази глава смятате ли, че са добри или лоши?

Решения и упътвания

1. Използвайте [Ctrl+Shift+F] в Eclipse и вижте разликите. След това отново с помощта на Eclipse преименувайте променливите, премахнете излишните оператори и променливи и направете текста, който се отпечатва на екрана по-смислен.
2. Внимателно следвайте препоръките за конструиране на качествен програмен код от настоящата тема.

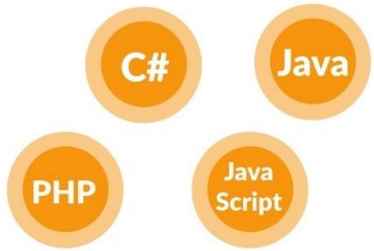
3. Вземете като пример някой качествено написан софтуер, примерно сорс кода на проекта Apache Ant (можете да го изтеглите от адрес http://ant.apache.org/srcdownload.cgi). Ще откриете много проблеми, свързани с форматирането, тъй като авторите на този проект имат друго мнение как трябва да се форматира сорс кода.
4. Кодът от стандартната библиотека на Java е писан от инженери с дългогодишен опит и в него рядко ще срещнете некачествен код. Въпреки всичко се срещат недоразумения като използване на сложни изрази, неправилно именувани променливи и други. Би трябвало да откриете проблеми с форматирането и неспазване на JavaBeans спецификацията (особено в по-стари класове): **`Object.toString()`**, **`Object.hashCode()`** са само първите примери.
5. Напишете "Java code conventions" в любимата ви Интернет търсачка.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 22. Как да решаваме задачи по програмиране?

Автор

Светлин Наков

В тази тема...

В настоящата тема ще дискутираме един препоръчителен подход за решаване на задачи по програмиране и ще го илюстрираме нагледно с реални примери. Ще дискутираме инженерните принципи, които трябва да следваме при решаването на задачи (които важат в голяма степен и за задачи по математика, физика и други дисциплини) и ще ги покажем в действие. Ще опишем стъпките, през които преминаваме при решаването на няколко примерни задачи и ще демонстрираме какви грешки се получават, ако не следваме тези стъпки. Ще обърнем внимание на някои важни стъпки от решаването на задачи (като например тестване), които обикновено се пропускат. Надяваме се да ви успеем да ви докажем чрез много примери, че за решаването на задачи по програмиране си има "рецепта" и да ви убедим колко много помага тя.

Основни принципи при решаване на задачи по програмиране

Сигурно си мислите, че сега ще ви напълним главата с празни приказки в стил "първо мисли, след това пиши" или "внимавайте като пишете, че да не пропуснете нещо". Всъщност тази тема няма да е толкова досадна и ще ви даде практически насоки как да подхождате при решаването на задачи, независимо дали са алгоритмични или други.

Без да претендираме за изчерпателност, ще ви дадем няколко важни препоръки, базирани на опита на Светлин Наков, който повече от 10 години подред е участвал редовно по български и международни състезания по програмиране, а след това е обучавал на програмиране и решаване на задачи студенти в Софийски университет "Св. Климент Охридски" (ФМИ на СУ), Нов Български Университет (НБУ), Национална академия по разработка на софтуер (НАРС), Telerik Academy и Софтуерен университет (СофтУни).

Нека започнем с първата важна препоръка.

Използвайте лист и химикал!

Захващането на лист и химикал и скицирането на примери и разсъждения по дадения проблем е нещо съвсем нормално и естествено – нещо, което всеки опитен математик, физик или софтуерен инженер прави, когато му поставят нетривиална задача.

За съжаление, от опита си с обучението на софтуерни инженери можем да споделим, че повечето начинаещи програмисти въобще не си носят лист и химикал. Те имат погрешното съзнание, че за да решават задачи по програмиране им е достатъчна само клавиатурата. На повечето им трябват доста време и провали по изпитите, за да достигат до важния извод, че използването на някаква форма на чертеж, скица или визуализация на проблема е от решаваща полза за неговото решаване.

Който не ползва лист и химикал, ще бъде силно затруднен при решаването на задачи по програмиране. Винаги скицирайте идеите си на хартия или на дъската!

Наистина, изглежда старомодно, но ерата на хартията все още не е отминала! Най-лесният начин човек да си скицира идеите и разсъжденията е като хване лист и химикал, а без да скицирате идеите си, е много трудно да разсъждавате.

Помислете например колко усилия ви трябват, за да умножавате петцифрени числа на ум и колко по-малко са усилията, ако имате лист и химикал (изключваме възможността да използваме електронни устройства). По същия начин е със задачите – когато трябва да измислите решение, ви трябва хартия да си драскате. Когато трябва да проверите дали решението ви е вярно, ви трябва отново хартия, да си разпишете един пример. Когато трябва да измисляте случаи, които вашето решение изпуска, отново ви трябва нещо, на което да си разписвате и драскате примери и идеи. Затова ползвайте лист и химикал!

Измислете идеи и ги пробвайте!

Решаването на дадена задача винаги започва от скицирането на някакъв пример върху лист хартия. Когато имате конкретен пример, можете да разсъждавате, а когато разсъждавате, ви хрумват идеи за решение на задачата.

Когато вече имате идея, ви трябват още примери, за да проверите дали идеята е добра. Тогава можете да нарисувате още няколко примера на хартия и да пробвате вашата идея върху тях. Уверете се, че идеята ви е вярна. Проследете идеята стъпка по стъпка, така, както ще я изпълни евентуална компютърна програма и вижте дали няма някакви проблеми.

Опитайте се да "счупите" вашата идея за решение – да измислите пример, при който не работи (контра-пример). Ако не успеете, вероятно сте на прав път. Ако успеете, помислете как да се справите с неработещия пример: измислете "поправка" на вашата идея за алгоритъм или измислете напълно нова идея.

За всичко това ви трябват лист, химикал и примери, които да измисляте и да си ги рисувате, след което да си пробвате върху тях различните идеи, които ви хрумват.

Решаването на задачи по програмиране започва от измисляне на идеи и проверяването им. Това става най-лесно като хванете лист и химикал и скицирате разсъжденията си. Винаги проверявайте идеите си с подходящи примери!

Горните препоръки са много полезни и в още един случай: когато сте на интервю за работа. Всеки опитен интервюиращ може да потвърди, че когато даде алгоритмична задача на кандидат за работа, очаква от него да хване лист и химикал и да разсъждава на глас като предлага различни идеи, които му хрумват. Хващането на лист и химикал на интервю за работа дава признаци за мислене и правилен подход за решаване на проблеми. Разсъждаването на глас показва, че можете да мислите. Дори и да не стигнете до правилно решение подходът към решаване на задачи ще направи добро впечатление на интервюиращия!

Разбивайте задачата на подзадачи!

Сложните задачи винаги могат да се разделят на няколко по-прости. Ще ви покажем това в примерите след малко. Нищо сложно на този свят не е направено наведнъж. Рецептата за решаване на сложни задачи е да се разбият логически на няколко по-прости (по възможност максимално независими една от друга). Ако и те се окажат сложни, можем да разбием и тях на няколко по-прости. Тази техника е известна като "разделяй и владей" и е използвана още от Римската империя.

Звучи просто на теория, но на практика не винаги е лесно да се направи. Тънкостта на решаване на алгоритмични задачи се крие в това да овладеете добре техниката на разбиването на задачата на по-прости подзадачи и, разбира се, да се научите да ви хрумват добри идеи, което става с много, много практика.

Сложните проблеми винаги могат да се разделят на няколко по-прости. Когато решавате задачи, разделяйте сложната задача на по-прости, задачи, които могат да се решат самостоятелно.

Разбъркване на тесте карти – пример

Нека дадем един пример: трябва да разбъркаме тесте карти в случаен ред. Да приемем, че тестето е дадено като масив или списък от N на брой обекти (всяка карта е обект). Това е задача, която изисква много стъпки (серия изваждания, вмъквания, размествания или преподреждания на карти). Тези стъпки сами по себе си са по-прости и по-лесни за реализация, отколкото цялостната задача за разбъркване на картите. Ако намерим начин да разбием сложната задача на множество простички стъпки, значи сме намерили начин да я решим. Именно в това се състои алгоритмичното мислене: в умението да разбиваме сложен проблем на серия по-прости проблеми, за които можем да намерим решение. Това, разбира се, важи не само за програмирането, но и за решаването на задачи по математика, геометрия, физика и други дисциплини. Точно алгоритмичното мислене е причината математиците и физиците много бързо да напредват, когато се захванат с програмиране.

Нека сега се върнем на нашата задача и да помислим кои са елементарните действия, които са нужни, за да разбъркаме в случаен ред картите?

Ако хванем в ръка тесте карти или си го нарисуваме по някакъв начин на лист хартия (например като серия кутийки с по една карта във всяка от тях), веднага ще ни хрумне идеята, че е необходимо да направим някакви размествания или пренареждания на някои от картите.

Разсъждавайки в този дух установяваме, че трябва да направим повече от едно разместване на една или повече карти, защото, ако направим само едно разместване, получената подредба няма да е съвсем случайна. Следователно ни трябват много на брой по-прости операции за единични размествания.

Стигнахме до първото разделяне на задачата на подзадачи: трябват ни серия размествания и всяко разместване можем да разгледаме като по-проста задача, част от решението на по-сложната.

Първа подзадача: единично разместване

Как правим "единично разместване"? На този въпрос има стотици отговори, но можем да вземем първата идея, която ни хрумва. Ако е добра, ще я ползваме. Ако не е добра, ще измислим друга.

Ето каква може да е първата ни идея: ако имаме тесте карти, можем да се сетим да разделим тестето на две части по случаен начин и да разменим едната част с другата. Имаме ли идея за "единично разместване" на картите? Имаме. Остава да видим дали тази идея ще ни свърши работа.

Нека се върнем на началната задача: трябва да получим случайно размесено тестето карти, което ни е дадено като вход. Ако хванем тестето и много на брой пъти го разцепим на две и разменим получените две части, ще получим случайно размесване, нали? Изглежда нашата първа идея за "единично разместване" ще свърши работа.

Втора подзадача: избор на случайно число

Как избираме случаен начин за разцепване на тестето? Ако имаме N карти, ни трябва начин да изберем число между 1 и N-1, нали?

За да решим тази подзадача, ни трябва или външна помощ, или да знаем, че тази задача в Java е вече решена и можем да ползваме вградения генератор на случайни числа наготово.

Ако не се сетим да потърсим в Интернет как в Java се генерират случайни числа, можем да си измислим и наше собствено решение, например да въвеждаме един ред от клавиатурата и да измерваме интервала време между стартирането на програмата и натискането на [Enter] за край на въвеждането. Понеже при всяко въвеждане това време ще е различно (особено, ако можем да отчитаме с точност до наносекунди), ще имаме начин да получим случайно число. Остава въпросът как да го накараме да бъде в интервала от 1 до N-1, но вероятно ще се сетим да ползваме остатъка от деление на (N-1) и да си решим проблема.

Виждате, че дори простите задачи могат да имат свои подзадачи или може да се окаже, че за тях вече имаме готово решение. Когато намерим решение, приключваме с текущата подзадача и се връщаме към оригиналната задача, за да търсим идеи и за нейното решаване. Нека направим това.

Трета подзадача: комбиниране на разместванията

Да се върнем пак на началната задача. Чрез последователни разсъждения стигнахме до идеята много пъти да извършим операцията "единично разместване" в тестето карти докато тестето се размести добре. Това изглежда коректно и можем да го пробваме.

Сега възниква въпросът колко пъти да извършим операцията "единично разместване". 100 пъти достатъчно ли е? А не е ли много? А 5 пъти достатъчно ли е, не е ли малко? За да дадем добър отговор на този въпрос трябва да помислим малко. Колко карти имаме? Ако картите са малко, ще ни трябват малко размествания. Ако картите са много, ще ни трябват повече размествания, нали? Следователно броят размествания изглежда зависи от броя карти.

За да видим колко точно трябва да са тези размествания, можем да вземем един пример. Да вземем стандартно тесте карти. Колко карти има в него? Всеки картоиграч ще каже, че са 52. Ами тогава да помислим колко разцепвания на тестето на две и разменяния на двете половини ни трябват, за да разбъркаме случайно 52 карти. Дали 52 е добре? Ако направим 52 "единични размествания" изглежда, че ще е достатъчно, защото заради случайния избор ще сцепим средно по 1 път между всеки две карти (това е видно и без да четем дебели книги по вероятности и статистика). А дали 52 не е много? Можем да измислим и по-малко число, което ще е достатъчно,

примерно половината на 52. Това също изглежда достатъчно, но ще е по-трудно да се обосновем защо.

Някои биха тръгнали с дебелите формули от теорията на вероятностите, но има ли смисъл? Числото 52 не е ли достатъчно малко, за да търсим по-малко. Цикъл от 1 до 52 минава мигновенно, нали? Картите няма да са един милиард, нали? Следователно няма нужда да мислим в тази посока. Приемаме, че правим толкова "единични разместванията", колкото са картите и това хем е достатъчно, хем не е прекалено много. Край, тази подзадача е решена.

Още един пример: сортиране на числа

Нека разгледаме накратко и още един пример. Даден е масив с числа и трябва да го сортираме по големина, т.е. да подредим елементите му в нарастващ ред. Това е задача, която има десетки концептуално различни методи за решаване и вие можете да измислите стотици идеи, някои, от които са верни, а други – не съвсем.

Ако имаме тази задача и приемем, че е забранено да се ползват вградените в Java класове за сортиране, е нормално да вземем лист и химикал, да си направим един пример и да започнем да разсъждаваме. Можем да достигнем до много различни идеи, примерно:

- Можем да изберем най-малкото число, да го отпечатаме и да го изтрием от масива. След това можем да повторим същото многократно докато масивът свърши. Разсъждавайки по тази идея можем да разделим задачата на няколко по-прости задачки: намиране на най-малко число в масив; изтриване на число от масив; отпечатване на число.
- Можем да вземем най-малкото число и да го преместим най-отпред (чрез изтриване и вмъкване). След това в останалата част от масива можем пак да намерим най-малкото число и да го преместим веднага след първото. На k-тата стъпка ще имаме първите k най-малки числа в началото на масива. При този подход задачата се разделя по естествен начин на няколко по-малки задачки: намиране на най-малко число в част от масив и преместване на число от една позиция на масив в друга. Последната задачка може да се разбие на две по-малки: "изтриване от масив" и "вмъкване в масив").
- Можем да подходим и коренно различно: да разделим масива на две части с равен брой елементи, след което да сортираме първата част, да сортираме втората част и накрая да обединим двете части. Можем да приложим същото рекурсивно за всяка от частите докато не достигнем до част с големина един елемент, който очевидно е сортиран. При този подход имаме пак разделяне на сложната задача на няколко по-прости подзадачи: разделяне на масив на две равни (или почти равни) части; сливане на сортирани масиви.

Няма да продължаваме повече. Всеки може да измисли още много идеи за решаване на задачата или да ги прочете в някоя книга по алгоритми. Показахме ви, че винаги сложната задача може да се раздели на няколко по-малки и по-прости задачки. Това е правилният подход при решаване на задачи по програмиране – да мислим за големия проблем като за съвкупност от няколко по-малки проблема. Това е техника, която се усвоява бавно с времето, но рано или късно ще трябва да свикнете с нея.

Проверете идеите си!

Изглежда не остана нищо повече за измисляне. Имаме идея. Тя изглежда, че работи. Остава да проверим дали наистина работи или само така си мислим и да след това да се ориентираме към имплементация.

Как да проверим идеята си? Обикновено това става с някакъв пример или с няколко примера. Трябва да подберете такива, примери, които в пълнота покриват различните случаи, които вашия алгоритъм трябва да преодолее. Примерите трябва хем да не са лесни за вашия алгоритъм, хем да са достатъчно прости, за да ги разпишете бързо и лесно. Такива примери наричаме "добри представители на общия случай".

Например, ако реализираме алгоритъм за сортиране на масив в нарастващ ред, удачно е да вземем пример с 5-6 числа, сред които има 2 еднакви, а останалите са различни. Числата трябва първоначално да са подредени в случаен ред. Това е добър пример, понеже покрива много голяма част от случаите, в които вашия алгоритъм трябва да работи.

За същата задача са сортиране има множество неподходящи примери, с които няма да можете ефективно да проверите дали вашата идея за решение работи коректно. Например можем да вземем пример само с 2 числа. За него алгоритъмът може да работи, но по идея да е грешен. Можем да вземем пример само с еднакви числа. При него всеки алгоритъм за сортиране ще работи. Можем да вземем пример с числа, които са предварително подредени по големина. И за него алгоритъмът може да работи, но да е грешен.

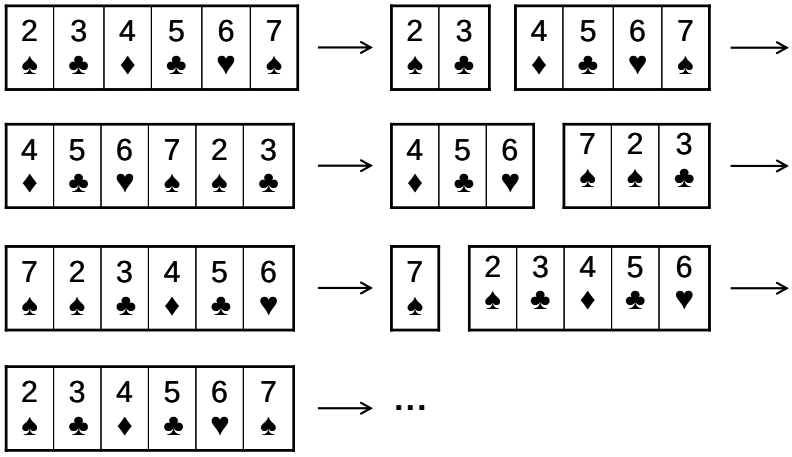
Когато проверявате идеите си подбирайте подходящи примери. Те трябва хем да са прости и лесни за разписване, хем да не са частен случай, при който вашата идея би могла да работи, но да е грешна в общия случай. Примерите, които избирате, трябва да са добри представители на общия случай – да покриват възможно повече случаи, без да са големи и сложни.

Разбъркване на карти: проверка на идеята

Нека измислим един пример за нашата задача за разбъркване на карти, да кажем с 6 карти. За да е добър примера, картите не трябва да са малко (да

кажем 2-3), защото така примерът е прекалено лесен, но не трябва и да са много, за да можем бързо да проиграем нашата идея върху примера. Добре е картите да са подредени първоначално по големина или даже за по-лесно да са поредни, за да може накрая лесно да видим дали са разбъркани – ако се запазят поредни или частично подредени, значи разбъркването не работи добре. Може би е най-хитро да вземем 6 карти, които са поредни, без значение на боята.

Вече измислихме пример, който е добър представител на общия случай за нашата задача. Нека да го нарисуваме на лист хартия и да проиграем върху него измисления алгоритъм. Трябва 6 пъти подред да сцепим на случайно място поредицата карти и да разменим получените 2 части. Нека картите първоначално са наредени по големина. Очакваме накрая картите да са случайно разбъркани. Да видим какво ще получим:



Няма нужда да правим 6 разцепвания. Вижда се, че след 3 разместванията се върнахме в изходна позиция. Това едва ли е случайно. Какво стана? Открихме проблем в алгоритъма. Изглежда, че нашата идея е грешна. Като се замислим малко, се вижда, че всяко единично разместване през случайната позиция k всъщност ротира наляво тестето карти k пъти и след общо N ротации стигаме до изходна позиция. Добре, че тествахме на ръка алгоритъма преди да сме написали програмата, нали?

При проблем измислете нова идея!

Нормално е, след като намерим проблем в нашата идея, да измислим нова идея, която би трябвало да работи. Това може да стане по два начина: или да поправим старата си идея, като отстраним дефектите в нея, или да измислим напълно нова идея. Нека видим как това работи за нашата задача за разбъркване на карти.

Измислянето на решение на задача по програмиране е итеративен процес, който включва последователно

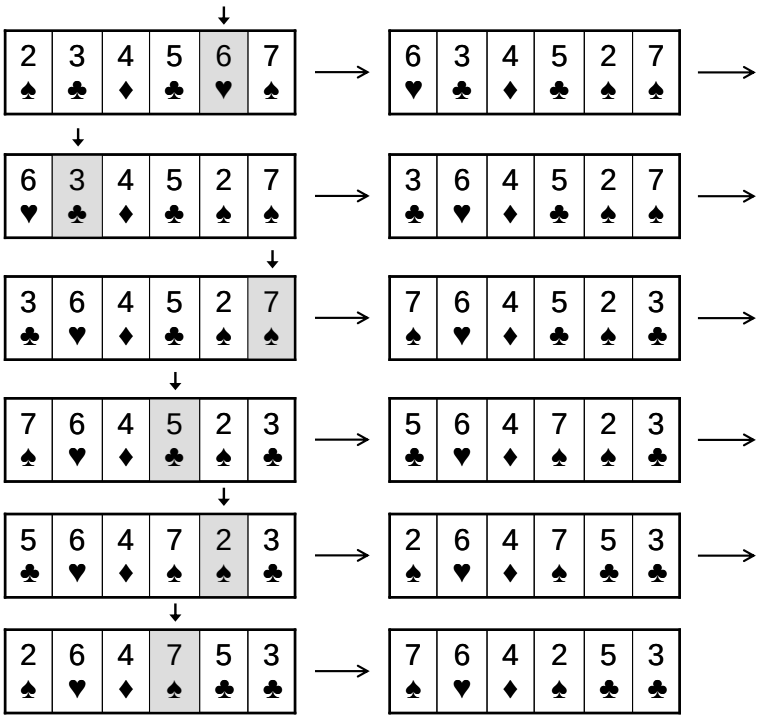
	измисляне на идеи, изпробването им и евентуално замяната им с по-добри идеи при откриване на проблем. Понякога още първата идея е правилна, а понякога пробваме и отхвърляме една по една много различни идеи докато стигнем до идея, която да ни свърши работа.

Да се върнем на нашата задача. Първото нещо, което ни хрумва, е да видим защо е грешна нашата първа идея и да се опитаме да я поправим, ако това е възможно. Проблемът лесно се забелязва: последователното разцепване на тестето на две части и размяната им не води до случайна наредба на картите, а до някаква тяхна ротация (изместване наляво с някакъв брой позиции).

Как да поправим алгоритъма? Необходим ни е по-умен начин да правим единичното разместване, нали? Хрумва ни следната идея: взимаме две случайни карти и ги разменяме една с друга? Ако го направим N на брой пъти, сигурно ще се получи случайна наредба. Идеята изглежда по-добра от предната и може би работи. Вече знаем, че преди да мислим за реализация на новия алгоритъм трябва да го проверим. Започваме да скицираме на хартия какво ще се случи за нашия пример с 6 карти.

В този момент ни хрумва нова като че ли по-добра идея. Не е ли по-лесно на всяка стъпка да вземем случайна карта и да я разместим с първата? Изглежда по-просто и по-лесно за реализация, а резултатът би трябвало пак да е случаен. Първоначално ще разменим карта от случайна позиция k_1 с първата карта. Ще имаме случайна карта на първа позиция и първата карта ще бъде на позиция k_1. На следващата стъпка ще изберем случайна карта на позиция k_2 и ще я разменим с първата карта (картата от позиция k_1). Така вече първата карта си е сменила позицията, картата от позиция k_1 си е сменила позицията и картата от позиция k_2 също си е сменила позицията. Изглежда, че на всяка стъпка по една карта си сменя позицията със случайна. След такива N стъпки можем да очакваме всяка карта средно по веднъж да си е сменила мястото и следователно картите би трябвало да са добре разбъркани.

Дали това наистина е наистина така? Да не стане като предния път? Нека проверим. Отново ще вземем 6 карти, които представляват добре подбран пример за нашата задача (добър представител на общия случай). Ето какво се получава:



От примера виждаме, че резултатът е правилен – получава се наистина случайно разбъркване на нашето примерно тесте от 6 карти. Щом нашият алгоритъм работи за 6 карти, би трябвало да работи и за друг брой. Ако не сме убедени в това, е хубаво да вземем друг пример, който изглежда, че е по-труден за нашия алгоритъм.

Ако сме твърдо убедени, че идеята е вярна, може и да си спестим разписването на повече примери на хартия. Можем да продължим напред с решаването на задачата.

Да обобщим какво направихме до момента и как чрез последователни разсъждения стигнахме до идея за решаването на задачата. Следвайки всички препоръки, изложени до момента, минахме през следните стъпки:

- Използвахме лист и химикал, за да си скицираме тесте карти за разбъркване. Нарисувахме си последователност от кутийки на лист хартия и така успяхме визуално да си представим картите.
- Имайки визуална представа за проблема, ни хрумнаха някои идеи: първо, че трябва да правим някакви единични разместания и второ, че трябва да ги правим много на брой пъти.
- Решихме да правим единични разместания чрез цепене на картите на случайно място и размяна на двете половини.
- Решихме, че трябва да правим толкова разместания, колкото са картите в тестето.

- Сблъскахме се и с проблема за избор на случайно число, но избрахме решение наготово.
- Разбихме оригиналната задача на три подзадачи: единично разместване; избор на случайно число; комбиниране на единичните разместванията.
- Проверихме дали идеята работи и намерихме грешка. Добре, че направихме проверка преди да напишем кода!
- Измислихме нова стратегия за единично разместване, която изглежда по-надеждна.
- Проверихме новата идея с подходящи примери и имаме увереност, че е правилна.

Вече имаме идея за решение на задачата и тя е проверена с примери. Това е най-важното за решаването на една задача – да измислим алгоритъма. Остава по-лесното – да реализираме идеята си. Нека видим как става това.

Подберете структурите от данни!

Ако вече имаме идея за решение, която изглежда правилна и е проверена с няколко примера, остава да напишем програмния код, нали? Какво изпуснахме? Измислихме ли всичко необходимо, за да можем бързо, лесно и безпроблемно да напишем програма, която реализира нашата идея за решаване на задачата?

Това, което изпуснахме, е да си представим как нашата идея (която видяхме как работи на хартия) ще бъде имплементирана като компютърна програма. Това не винаги е елементарно и понякога изисква доста време и допълнителни идеи. Това е важна стъпка от решаването на задачи: да помислим за идеите си в термините на компютърното програмиране. Това означава да разсъждаваме с конкретни структури от данни, а не с абстракции като "карта" и "тесте карти". Трябва да подберем подходящи структури от данни, с които да реализираме идеите си.

Преди да преминете към имплементация на вашата идея помислете за структурите от данни. Може да се окаже, че вашата идея не е толкова добра, колкото изглежда. Може да се окаже, че е трудна за реализация или неефективна. По-добре да откриете това сега, отколкото по-късно.

В нашия случай говорихме за "размяна на случайна карта с друга", а в програмирането това означава да разместим два елемента в някаква структура от данни (примерно масив, списък или нещо друго). Стигнахме до момента, в който трябва да изберем структурите от данни и ще ви покажем как се прави това.

В каква структура да пазим тестето карти?

Първият въпрос, който възниква, е в каква структура от данни да съхраняваме тестето карти. Могат да ни хрумнат всякакви идеи, но не всички структури от данни са подходящи. Нека разсъждаваме малко по въпроса. Имаме съвкупност от карти и наредбата на картите в тази структура е от значение. Следователно трябва да използваме структура, която съхранява съвкупност от елементи и запазва наредбата им.

Можем ли да ползваме масив?

Първото, което можем да се сетим, е да използваме "масив". Това е най-простата структура за съхранение на съвкупност от елементи. Масивът може да съхранява съвкупност от елементи и в него елементите имат наредба (първи, втори трети и т.н.). Масивът не може да променя първоначално определения му размер.

Подходяща структура ли е масивът? За да си отговорим на този въпрос, трябва да помислим какво трябва да правим с тестето карти, записано в масив и да проверим дали всяка от необходимите ни операции може да се реализира ефективно с масив.

Кои са операциите с тестето карти, които ще ни се наложи да реализираме за нашия алгоритъм? Нека ги изброим:

- Избор на случайна карта. Понеже в масива имаме достъп до елементите по индекс, можем да изберем случайно място в него (вдясно от първата позиция) чрез избор на случайно число k в интервала от 1 до N-1.
- Размяна на карта на позиция k с първата карта (единично разместване). След като сме избрали случайна карта, трябва да я разменим с първата. И тази операция изглежда проста. Можем да направим размяната на три стъпки чрез временна променлива.
- Въвеждане на тестето / обхождане на картите от тестето / отпечатване на тестето – всички тези операции биха могли да ни потрябват, но изглежда тривиално да ги реализираме с масив.

Изглежда, че масивът може да ни свърши работа са съхранение на тесте карти.

Можем ли да ползваме друга структура?

Нормално е да си зададем въпроса дали масив е най-подходящата структура от данни за реализиране на операциите, които нашата програма трябва да извършва върху тестето карти. Изглежда, че всички операции могат лесно да се реализират с масив.

Все пак, нека помислим можем ли да изберем по-подходяща структура от масив. Нека помислим какви са възможностите ни:

- Свързан списък – нямаме директен достъп по номер на елемент и ще ни е трудно да избираме от списъка случайна карта.
- Статичен списък с променлива дължина (**ArrayList**) – изглежда, че притежава всички предимства на масивите и може да реализира всички операции, които ни трябват, по същия начин, както с масив. Печелим малко удобство – в **ArrayList** можем лесно да трием и добавяме, което може да улесни въвеждането на картите и някои други помощни операции.
- Стек / опашка – тестето карти няма поведение на FIFO / LIFO и следователно тези структури не са подходящи.
- Множество (**TreeSet** / **HashSet**) – в множествата няма наредба и това е съществена пречка, за да ги използваме.
- Хеш-таблица – структурата "тесте карти" не е от вида ключ-стойност и следователно хеш-таблицата не може да го съхранява и обработва ефективно. Освен това хеш-таблиците не запазват подредбата на елементите.

Общо взето изчерпахме основните структури от данни, които съхраняват и обработват съвкупности от елементи и стигнахме до извода, че масив или **ArrayList** ще ни свършат работа, а **ArrayList** е по-гъвкав и удобен от обикновения масив. Взимаме решение да ползваме **ArrayList** за съхранението и обработката на тестето карти.

Изборът на структура данни започва с изброяване на ключовите операции, които ще се извършват върху нея. След това се анализират възможните структури, които могат да бъдат използвани и от тях се избира тази, която най-лесно и ефективно реализира тези операции. Понякога се прави компромис между леснота на реализация и ефективност.

Как да пазим другите информационни обекти?

След като решихме първия проблем, а именно как да представяме в паметта тесте от карти, следва да помислим дали има и други обекти, с които боравим, за които следва да помислим как да ги представяме. Като се замислим, освен обектите "карта" и "тесте карти", нашият алгоритъм не използва други информационни обекти.

Възниква въпросът как да представим една карта? Можем да я представим като символен низ, като число или като клас с две полета – лице и боя. Има, разбира се и други варианти, които имат своите предимства и недостатъци.

Преди да навлезем в дълбоки разсъждения кое представяне е най-добро, нека се върнем на условието на задачата. То предполага, че тестето карти ни е дадено (като масив или списък) и трябва да го разместим. Какво точно

представлява една карта няма никакво значение за тази задача. Дори няма значение дали разместваме карти за игра, фигури за шах, кашони с домати или някакви други обекти. Имаме наредена последователност от обекти и трябва да я разбъркаме. Фактът, че разбъркваме карти, няма значение за нашата задача и няма нужда да губим време да мислим как точно да представим една карта. Нека просто се спрем на първата идея, която ни хрумва, примерно да си дефинираме клас Card с полета face и suit. Дори да изберем друго представяне (примерно число от 1 до 52), това не е съществено. Няма да дискутираме повече този въпрос.

Сортиране на числа – подбор на структурите данни

Преди да продължим нататък, нека разгледаме още един пример, при който имаме нужда от избор на структури от данни. Нека имаме задачата за сортиране по големина на съвкупност от числа. Нека сме избрали да използваме най-простия алгоритъм, за който сме се сетили: да взимаме докато може най-малкото число, да го отпечатваме и да го изтриваме. Тази идея лесно се разписва на хартия и лесно се убеждаваме, че е коректна.

Каква структура от данни да ползваме за съхранение на числата? Отново, за да си отговорим на този въпрос, е необходимо помислим какви операции имаме да извършваме върху тези числа. Операциите са следните:

- Търсене на най-малка стойност в структурата.
- Изтриване на намерената най-малка стойност от структурата.

Очевидно използването на масив не е разумно, защото не разполагаме с операцията "изтриване". Използването на **`ArrayList`** изглежда по-добре, защото и двете операции можем да реализираме сравнително просто и лесно. Структури като стек и опашка няма да ни помогнат, защото нямаме LIFO или FIFO поведение. От хеш-таблица няма особен смисъл, защото в нея няма бърз начин за намиране на най-малка стойност, въпреки че изтриването на елемент би могло да е по-ефективно.

Стигаме до структурите **`HashSet`** и **`TreeSet`**. Множествата имат проблема, че не поддържат възможност за съхранение на еднакви елементи. Въпреки това, нека ги разгледаме. Структурата **`HashSet`** не представлява интерес, защото при нея отново нямаме лесен начин да намерим най-малкия елемент. Обаче структурата **`TreeSet`** изглежда обещаваща. Нека я разгледаме.

Класът **`TreeSet`** според документацията на Java държи елементите си в балансирано дърво и поддържа операцията "изваждане на най-малкия елемент". Колко интересно! Хрумва ни нова идея: вкарваме всички елементи в **`TreeSet`** и изкарваме от него итеративно най-малкия елемент докато елементите свършат. Просто, лесно и ефективно. Имаме наготово двете операции, които ни интересуват (търсене на най-малък елемент и изтриването му от структурата) в методите **`first()`** и **`remove()`**.

Докато си представяме конкретната имплементация и се ровим в документацията прочитаме нещо още по-интересно: класът **`TreeSet`** държи вътрешно елементите си подредени по големина. Ами нали това се иска в задачата: да наредим елементите по големина. Следователно, ако ги вкараме в **`TreeSet`** и след това обходим елементите му (чрез неговия итератор), те ще бъдат подредени по големина. Задачата е решена.

Докато се радваме, се сещаме за един забравен проблем: **`TreeSet`** не поддържа еднакви елементи, т.е. ако имаме числото 5 няколко пъти, то ще се появи в множеството само веднъж. В крайна сметка при сортирането ще загубим безвъзвратно някои от елементите.

Естествено е да потърсим решение на този проблем. Ако има начин да пазим колко пъти се среща всеки елементи от множеството, това ще ни реши проблема. Тогава се сещаме за класа **`TreeMap`**. Той съхранява множество ключове, които са подредени по големина и във всеки ключ можем да имаме стойност. В стойността можем да съхраняваме колко пъти се среща даден елемент. Изглежда това решава проблема ни и можем да го реализираме, макар и не толкова лесно, колкото с **`ArrayList`** или с **`TreeSet`**.

Ако прочетем внимателно документацията за **`TreeMap`**, ще видим, че този клас вътрешно използва черно-червено дърво (каквото и да е това) и може някой ден да се досетим, че неусетно чрез разсъждения сме достигнали до добре известния алгоритъм "сортиране чрез дърво" (http://en.wikipedia.org/wiki/Binary_tree_sort).

Видяхте до какви идеи ви довеждат разсъжденията за избор на подходящи структури от данни за имплементация на вашите идеи. Тръгвате от един алгоритъм и неусетно измисляте нов, по-добър. Това е нормално да се случи в процеса на обмисляне на алгоритъма и е добре да се случи в този момент, а не едва когато сте написали вече 300 реда код, който ще се наложи да преправяте. Това е още едно доказателство, че трябва да помислите за структурите от данни преди да почнете да пишете кода.

Помислете за ефективността!

За пореден път изглежда, че най-сетне сме готови да хванем клавиатурата и да напишем кода на програмата. И за пореден път е добре да не бързаме. Причината е, че не сме помислили за нещо много важно: ефективност и бързодействие.

За ефективността трябва да се помисли още преди да се напише първи ред програмен код. Иначе рискувате да загубите много време за реализация на идея, която не върши работа.

Имаме идея за решаване на задачата (измислили сме алгоритъм). Идеята изглежда коректна (пробвали сме я с примери). Идеята изглежда, че може да се реализира (ще ползваме **`ArrayList`** за тестето карти и клас **`Card`** за

представянето на една карта). Обаче, нека помислим колко карти ще разбъркваме и дали избраната идея, реализирана с избраните структури от данни, ще работи достатъчно бързо.

Как оценяваме бързината на даден алгоритъм?

Бърз ли е нашият алгоритъм? За да си отговорим на този въпрос, нека помислим колко операции извършва той за разбъркването на стандартно тесте от 52 карти.

За 52 карти нашият алгоритъм прави 52 единични размествания, нали така? Колко елементарни операции отнема едно единично разместване? Операциите са 4: избор на случайна карта; запазване на първата карта във временна променлива; запис на случайната карта на мястото на първата; запис на първата карта (от временната променлива) на мястото, където е била случайната карта. Колко операции прави общо нашият алгоритъм за 52 карти? Операциите са приблизително 52 * 4 = 208.

Много операции ли са 208? Замислете се колко време отнема да завъртите цикъл от 1 до 208. Много ли е? Пробвайте! Ще се убедите, че цикъл от 1 до 1 000 000 при съвременните компютри минава неусетно бързо, а цикъл до 208 отнема смешно малко време. Следователно нямаме проблем с производителността. Нашия алгоритъм ще работи бързо за 52 карти.

Въпреки, че в реалността рядко играем с повече от 1 или 2 тестета карти, нека се замислим колко време ще отнеме да разбъркаме голям брой карти, да кажем 50 000? Ще имаме 50 000 единични размествания по 4 операции за всяко от тях или общо 200 000 операции, които ще се изпълнят на момента, без да се усети каквото и да е забавяне.

Ефективността е въпрос на компромис

В крайна сметка правим извода, че алгоритъмът, който сме измислили е ефективен и ще работи добре дори при голям брой карти. Имахме късмет. Обикновено нещата не са толкова прости и трябва да се прави компромис между бързодействие на алгоритъма и усилията, които влагаме, за да го измислим и имплементираме. Например, ако сортираме числа, можем да го направим за 5 минути с първия алгоритъм, за който се сетим, но можем да го направим и много по-ефективно, за което ще употребим много повече време (да търсим и да четем из дебелите книги и в Интернет). В този момент трябва да се прецени струва ли си усилията. Ако ще сортираме 20 числа, няма значене как ще го направим, все ще е бързо, дори с най-глупавия алгоритъм. Ако сортираме 20 000 числа вече алгоритъмът има значение, а ако сортираме 20 000 000 числа, задачата придобива съвсем друг характер. Времето, необходимо да реализираме ефективно сортиране на 20 000 000 числа е далеч повече от времето да сортираме 20 числа, така че трябва да помислим струва ли си.

Ефективността е въпрос на компромис – понякога не си струва да усложняваме алгоритъма и да влагаме време и усилия, за да го направим по-бърз, а друг път бързината е ключова изискване и трябва да й обърнем сериозно внимание.

Сортиране на числа – оценяване на ефективността

Видяхте, че подхода към въпроса с ефективността силно зависи от изискванията за бързодействие. Нека се върнем сега на задачата за сортирането на числа, защото искаме да ви покажем, че ефективността е пряко свързана с избора на структури от данни.

Да се върнем отново на въпроса за избор на структура от данни за съхранение на числата, които трябва да сортираме по големина в нарастващ ред. Дали да изберем **`ArrayList`** или **`TreeMap`**? Не е ли по-добре да ползваме някаква проста структура, която добре познаваме, отколкото някоя сложна, която изглежда, че ще ни свърши работата малко по-добре. Вие познавате ли добре черно-червените дървета (вътрешната имплементация на **`TreeMap`**)? С какво са по-добри от **`ArrayList`**? Всъщност може да се окаже, че няма нужда да си отговаряте на този въпрос.

Ако трябва да сортирате 20 числа, има ли значение как ще го направите? Взимате първия алгоритъм, за който се сетите, взимате първата структура от данни, която изглежда, че ще ви свърши работа и готово. Няма никакво значение колко са бързи, защото числата са изключително малко.

Ако, обаче трябва да сортирате 300 000 числа, нещата са съвсем различни. Тогава ще трябва внимателно да проучите как работи класът **`TreeMap`** и колко бързо става добавянето и търсенето в него, след което ще трябва да оцените ориентировъчно колко операции ще са нужни за 300 000 добавяния на число и след това колко още операции ще отнеме обхождането. Ще трябва да прочетете документацията, където пише, че добавянето отнема средно $\log_2(N)$ операции, където N е броят елементи в структурата. Чрез дълги и мъчителни сметки (за които ви трябват допълнителни умения) може да оцените грубо, че ще са необходими около 5-6 милиона стъпки за цялото сортиране, което е приемливо бързо.

По аналогичен път, можете да се убедите, че търсенето и изтриването в **`ArrayList`** с N елемента отнема N стъпки и следователно за 300 000 елемента ще ни трябват приблизително 2 * 300 000 * 300 000 стъпки! Всъщност това число е силно закръглено нагоре, защото в началото нямате 300 000 числа, а само 1, но грубата оценка е пак приблизително вярна. Получава се екстремално голям брой стъпки и простичкият алгоритъм няма да работи за такъв голям брой елементи (програмата мъчително ще увисне).

Отново стигаме до въпроса с компромиса между сложния и простия алгоритъм. Единият е по-лесен за имплементиране, но е по-бавен. Другият

е по-ефективен, но е по-сложен за имплементиране и изисква да четем документация и дебели книги, за да разберем колко бързо ще работи. Въпрос на компромис.

Имплементирайте алгоритъма си!

Най-сетне стигаме до имплементация на нашата идея за решаване на задачата. Вече имаме работеща и проверена идея, подбрали сме подходящи структури от данни и остава да напишем кода. Ако не сме направили това, трябва да се върнем на предните стъпки.

Ако нямате измислена идея за решение, не почвайте да пишете код! Какво ще напишете, като нямате идея за решаване на задачата? Все едно да отидете на гарата и да се качите на някой влак, без да сте решили за къде ще пътувате.

Типично за начинаещите програмисти, е като видят задачата да почнат веднага да пишат и след като загубят няколко часа в писане на необмислени идеи (които им хрумват докато пишат), да се сетят да помислят малко. Това е грешно и целта на всички препоръки до момента е да ви предпази от такъв лекомислен и крайно неефективен подход.

Ако не сте проверили дали идеите ви са верни, не почвайте да пишете код! Трябва ли да напишете 300 реда код и тогава да откриете, че идеята ви е тотално сбъркана и трябва да почнете отначало?

Писането на кода при вече измислена и проверена идея изглежда просто и лесно, но и за него се изискват специфични умения и най-вече опит. Колкото повече програмен код сте писали, толкова по-бързо, ефективно и без грешки се научавате да пишете. С много практика ще постигнете лекота при писането и постепенно с времето ще се научите да пишете не само бързо, но и качествено. За качеството на кода можете да прочетете в главата "Качествен програмен код", така че, нека се фокусираме върху правилния подход при писането на кода.

Считаме, че би трябвало вече да сте овладели начални техники, свързани с писането на програмен код: как да работите със средата за разработка (Eclipse), как да ползвате компилатора, как да разчитате грешките, които той ви дава, как да ползвате подсказките (auto complete), как да генерирате методи, конструктори и свойства, как да поправяте грешки и как да изпълнявате и дебъгвате програмата. Затова съветите, които следват, са свързани не със самото писане на програмни редове код, а с цялостния подход при имплементиране на алгоритми.

Пишете стъпка по стъпка!

Случвало ли ви се е да напишете 200-300 реда код, без да опитате поне веднъж да компилирате и да тествате дали нещо работи? Не правете така! Не пишете много код на един път, а вместо това пишете стъпка по стъпка.

Как да пишем стъпка по стъпка? Това зависи от конкретната задача и от начина, по който сме я разделили на подзадачи. Например, ако задачата се състои от 3 независими части, напишете първо едната част, компилирайте я, тествайте я с някакви примерни входни данни и след като се убедите, че работи, преминете към следващите части. След това напишете втората част, компилирайте я, тествайте я и когато е готова и тя, преминете към третата част. Когато сте написали и последната част и сте се убедили, че работи, преминете към обстойно тестване на цялата програма.

Защо да пишем на части? Когато пишете на части, стъпка по стъпка, вие намалявате обема код, над който се концентрирате във всеки един момент. По този начин намалявате сложността на проблема, като го решавате на части. Спомнете си: големият и сложен проблем винаги може да се раздели на няколко по-малки и по-прости проблема, за които лесно ще намерите решение.

Когато напишем голямо количество код, без да сме опитали да компилираме поне веднъж, се натрупват голямо количество грешки, които могат да се избегнат чрез просто компилиране. Съвременните среди за програмиране (като Eclipse) се опитват да откриват синтактичните грешки автоматично още докато пишете кода. Ползвайте тази възможност и отстранявайте грешките възможно най-рано. Ранното отстраняване на проблеми отнема по-малко време и нерви. Късното отстраняване на грешки и проблеми може да коства много усилия, дори понякога и цялостно пренаписване на програмата.

Когато напишете голямо количество код, без да го тествате и след това решите наведнъж да го изпробвате за някакви примерни входни данни, обикновено се натъквате на множество проблеми, изсипващи се един след друг, като колкото повече е кодът, толкова по-трудно е те да бъдат оправени. Проблемите могат да са причинени от необмислено използване на неподходящи структури от данни, грешен алгоритъм, необмислено структуриране на кода, грешно условие в **`if`**-конструкция, грешно организиран цикъл, излизане извън граници на масив и много други проблеми, които е можело да бъдат отстранени много по-рано и с много по-малко усилия. Затова не чакайте последния момент. Отстранявайте грешките възможно най-рано.

Пишете програмата на части, а не наведнъж. Напишете някаква логически отделена част, компилирайте я, отстранете грешките, тествайте я и когато тя работи, преминете към следващата част.

Писане стъпка по стъпка – пример

За да илюстрираме на практика как можем да пишем стъпка по стъпка, нека се захванем с имплементация на алгоритъма за разбъркване на карти, който измислихме следвайки препоръките за решаване на алгоритмични задачи, описани по-горе.

Стъпка 1 – Дефиниране на клас "карта"

Тъй като трябва да разбъркваме карти, можем да започнем с дефиницията на класа "карта", тъй като ако нямаме как да представяме една карта, няма да има и как да представяме тесте карти и няма да има как да дефинираме метода за разбъркване на картите. Вече споменахме, че представянето на картите е извън обхвата на поставената задача, така че всякакво представяне би ни свършило работа.

Ще дефинираме клас "карта" с полета лице и боя. Ще използваме символен низ за лицето (с възможни стойности "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q" или "K") и изброен тип за боята (с възможни стойности "спатия", "каро", "купа" или "пика"). Класът **Card** би могъл да изглежда по следния начин:

Card.java

```
public class Card {
   private String face;
   private Suit suit;

   public Card(String face, Suit suit) {
      this.face = face;
      this.suit = suit;
   }

   public String getFace() {
      return face;
   }

   public Suit getSuit() {
      return suit;
   }

   @Override
   public String toString() {
      String card = "(" + this.face + " " + this.suit +")";
      return card;
   }
}

enum Suit {
```

```
   CLUB, DIAMOND, HEART, SPADE
}
```

За удобство дефинирахме и метод **toString()** в класа **Card**, с който можем по-лесно да отпечатваме дадена карта на конзолата. За боите дефинирахме изброен тип **Suit**.

Изпробване на класа "карта"

Някои от вас биха продължили да пишат напред, но следвайки принципа "програмиране стъпка по стъпка", трябва първо да тестваме дали класа Card се компилира и работи правилно. За целта можем да си направим малка програмка, в която създаваме една карта и я отпечатваме:

TestCard.java

```
public class TestCard {
   public static void main(String[] args) {
      Card card = new Card("A", Suit.CLUB);
      System.out.println(card);
   }
}
```

Стартираме програмата и виждаме дали картата се е отпечатала коректно. Резултатът е следният:

```
(A CLUB)
```

Стъпка 2 – Създаване и отпечатване на тесте карти

Нека преди да преминем към същината на задачата (разбъркване на тесте карти по случаен ред) се опитаме да създадем тесте карти и да го отпечатаме. Така ще се убедим, че входът на метода за разбъркване на карти е коректен. Според направения анализ на структурите данни, трябва да използваме **ArrayList<Card>**, за да представяме тестето карти. Нека създадем тесте от 5 карти и да го отпечатаме:

CardsShuffle.java

```
import java.util.ArrayList;

public class CardsShuffle {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      cards.add(new Card("2", Suit.CLUB));
      cards.add(new Card("7", Suit.HEART));
      cards.add(new Card("A", Suit.SPADE));
```

```
        cards.add(new Card("J", Suit.CLUB));
        cards.add(new Card("10", Suit.DIAMOND));

        printCards(cards);
    }

    public static void printCards(ArrayList<Card> cards) {
        for (Card card : cards) {
            System.out.print(card);
        }
        System.out.println();
    }
}
```

Отпечатване на тестето – тестване на кода

Преди да продължим напред, стартираме програмата и проверяваме дали сме получили очаквания резултат. Изглежда, че няма грешки и резултатът е коректен:

```
(2 CLUB)(7 HEART)(A SPADE)(J CLUB)(10 DIAMOND)
```

Стъпка 3 – Единично разместване

Нека реализираме поредната стъпка от решаването на задачата – подзадачата за единично разместване. Когато имаме логически отделена част от програмата е добра идея да я реализираме като отделен метод. Да помислим какво приема методът като вход и какво връща като изход. Като вход би трябвало да приема тесте карти (**`ArrayList<Card>`**). В резултат от работата си методът би трябвало да промени подадения като вход **`ArrayList<Card>`**. Методът няма нужда да връща нищо, защото не създава нов **`ArrayList`** за резултата, а оперира върху вече създаден.

Какво име да дадем на метода? Според препоръките за работа с методи трябва да дадем "говорящо" име – такова, което описва с 1-2 думи какво прави метода. Подходящо за случая е името **`performSingleExchange`**. Името ясно описва какво прави методът: извършва единично разместване.

Нека първо дефинираме метода, а след това напишем тялото му. Това е добра практика, тъй като преди да започнем да реализираме даден метод трябва да сме наясно какво прави той, какви параметри приема, какъв резултат връща и как се казва. Ето как изглежда дефиницията на метода:

```
private static void performSingleExchange(
  ArrayList<Card> cards){
  // TODO: Implement the method body
}
```

Следва да напишем тялото на метода. Първо трябва да си припомним алгоритъма, а той беше следният: избираме случайно число k в интервала от 1 до дължината на масива минус 1 и разменяме първия с k-тия елемент. Изглежда просто, но как в Java получаваме случайно число в даден интервал?

Търсете в Google!

Когато се натъкнем на често срещан проблем, за който нямаме решение, но знаем, че много хора са се сблъсквали с него, най-лесният начин да се справим е да потърсим в Google. Трябва да формулираме по подходящ начин нашето търсене. В случая търсим примерен Java код, който връща случайно число в даден интервал. Можем да пробваме следното търсене:

```
java random number example
```

На първо място в резултатите излиза Java програмка, която използва класа **java.util.Random**, за да генерира случайно число. Вече имаме посока, в която да търсим решение – имаме стандартен клас **Random**.

След това можем да се опитаме да налучкаме как се ползва този клас (често пъти това отнема по-малко време, отколкото да четем документацията). Опитваме да намерим подходящ статичен метод за случайно число, но се оказва, че такъв няма. Създаваме инстанция и търсим метод, който да ни върне число в даден диапазон. Оказва се, че такъв няма. Има обаче метод **nextInt(n)**, който по дадено число **n** връща случайно число в интервала от **0** до **n-1** (това Eclipse ни го показва автоматично при разглеждане на методите при auto complete). На нас ни трябва число от **1** до **n-1**. Как да го получим? Ами можем да вземем число от **0** до **n-2** и да му прибавим единица, нали?

Да опитаме да напишем кода на целия метод. Получава се нещо такова:

```
private static void performSingleExchange(
    ArrayList<Card> cards){
   Random rand = new Random();
   int randomIndex = 1 + rand.nextInt(cards.size()-2);
   Card firstCard = cards.get(1);
   Card randomCard = cards.get(randomIndex);
   cards.set(1, randomCard);
   cards.set(randomIndex, firstCard);
}
```

Единично разместване – тестване на кода

Следва тестване на кода. Преди да продължим нататък, трябва да се убедим, че единичното разместване работи коректно. Нали не искаме да открием евентуален проблем, когато тестваме метода за разбъркване на цялото тесте? Искаме, ако има проблем, да го открием веднага, а ако няма

проблем, да се убедим в това. Действаме стъпка по стъпка – преди да започнем следващата стъпка, проверяваме дали текущата е реализирана коректно. За целта си правим малка тестова програмка, да кажем с три карти (2, 3 и 4):

```
public static void main(String[] args) {
   ArrayList<Card> cards = new ArrayList<Card>();
   cards.add(new Card("2", Suit.CLUB));
   cards.add(new Card("3", Suit.HEART));
   cards.add(new Card("4", Suit.SPADE));
   performSingleExchange(cards);
   printCards(cards);
}
```

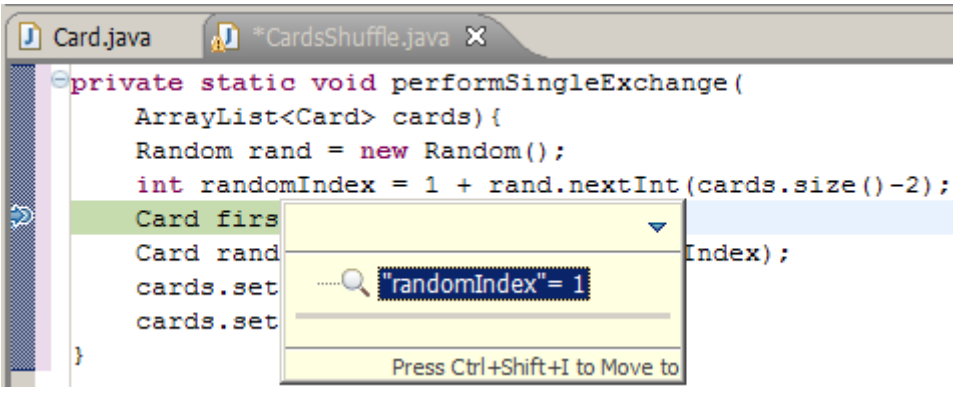
Нека изпълним няколко пъти единичното разместване с нашите 3 карти. Очакваме първата карта (двойката) да бъде разменена с някоя от другите две карти (с тройката или с четворката). Ако изпълним програмата много пъти, би следвало около половината от получените резултати да съдържат (3, 2, 4), а останалите – (4, 3, 2), нали така? Да видим какво ще получим. Стартираме програмата и получаваме следния резултат:

```
(2 CLUB)(3 HEART)(4 SPADE)
```

Ама как така? Какво стана? Да не съм забравил да изпълня единичното разместване преди да отпечатам картите? Има нещо гнило тук. Изглежда програмата не е направила нито едно разместване на нито една карта. Как стана тая работа?

Единично разместване – поправяне на грешките

Очевидно имаме грешка. Нека сложим точка на прекъсване и проследим какво се случва чрез дебъгера:



Видно е, че при първо стартиране случайната позиция се случва да има стойност 1. Това е допустимо, така че продължаваме напред. Като погледнем кода малко по-надолу, виждаме, че разменяме случайния

елемент с индекс 1 с елемента на позиция 1, т.е. със себе си. Очевидно нещо бъркаме. Сещаме се, че индексирането в Java започва от 0, не от 1, т.е. първият елемент е на позиция 0. Веднага поправяме кода:

```
private static void performSingleExchange(
    ArrayList<Card> cards){
   Random rand = new Random();
   int randomIndex = 1 + rand.nextInt(cards.size()-2);
   Card firstCard = cards.get(0);
   Card randomCard = cards.get(randomIndex);
   cards.set(0, randomCard);
   cards.set(randomIndex, firstCard);
}
```

Стартираме програмата няколко пъти и получаваме пак странен резултат:

```
(3 HEART)(2 CLUB)(4 SPADE)
(3 HEART)(2 CLUB)(4 SPADE)
(3 HEART)(2 CLUB)(4 SPADE)
```

Изглежда случайното число не е съвсем случайно. Какво пък има сега? Не бързайте да псувате виртуалната машина, Eclipse и всички други заподозрени виновници! Може би грешката е пак при нас. Да разгледаме извикването на метода **nextInt()**. Понеже **cards.size()** е 3, то винаги викаме **nextInt(3-2)**, т.е. **nextInt(1)**. Очакваме да ни върне число между 0 и 1 и като му прибавим единица, да получим случаен индекс между 1 и 2. Звучи коректно, обаче ако прочетем какво пише в документацията за метода **nextInt**, ще видим, че **nextInt(n)** връща число между 0 и n-1.

Имаме грешка с единица. Поправяме кода и се готвим за пореден път да тестваме дали работи. След втората поправка получаваме следната реализация на метода за единично разместване:

```
private static void performSingleExchange(
    ArrayList<Card> cards){
   Random rand = new Random();
   int randomIndex = 1 + rand.nextInt(cards.size()-1);
   Card firstCard = cards.get(0);
   Card randomCard = cards.get(randomIndex);
   cards.set(0, randomCard);
   cards.set(randomIndex, firstCard);
}
```

Ето какво би могло да се получи след няколко изпълнения:

```
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(4 SPADE)(3 HEART)(2 CLUB)
```

```
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(3 HEART)(2 CLUB)(4 SPADE)
```

Вижда се, че на мястото на първата карта отива всяка от следващите две карти, т.е. наистина имаме случайно разместване и всяка карта има еднакъв шанс да бъде избрана като случайна. Най-накрая сме готови с метода за единично разместване.

Стъпка 4 – Разместване на тестето

Последната стъпка е проста: прилагаме N пъти единичното разместване:

```
public static void shuffleCards(ArrayList<Card> cards) {
   for (int i=1; i<=cards.size(); i++) {
      performSingleExchange(cards);
   }
}
```

Ето как изглежда цялата програма:

CardsShuffle.java

```
import java.util.ArrayList;
import java.util.Random;

public class CardsShuffle {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      cards.add(new Card("2", Suit.CLUB));
      cards.add(new Card("7", Suit.HEART));
      cards.add(new Card("A", Suit.SPADE));
      cards.add(new Card("J", Suit.CLUB));
      cards.add(new Card("10", Suit.DIAMOND));

      System.out.println("Initial deck: ");
      printCards(cards);

      shuffleCards(cards);
      System.out.println("After shuffle: ");
      printCards(cards);
   }

   private static void performSingleExchange(
       ArrayList<Card> cards) {
      Random rand = new Random();
      int randomIndex = 1 + rand.nextInt(cards.size()-1);
      Card firstCard = cards.get(0);
```

```
      Card randomCard = cards.get(randomIndex);
      cards.set(0, randomCard);
      cards.set(randomIndex, firstCard);
   }

   public static void shuffleCards(ArrayList<Card> cards) {
      for (int i=1; i<=cards.size(); i++) {
         performSingleExchange(cards);
      }
   }

   public static void printCards(ArrayList<Card> cards) {
      for (Card card : cards) {
         System.out.print(card);
      }
      System.out.println();
   }
}
```

Разместване на тестето – тестване

Остава да пробваме дали целият алгоритъм работи – да го стартираме няколко пъти и да проверим дали всеки път се получава случайно разместване на картите. Ето какво се получава след няколко изпълнения на програмата:

```
(A SPADE)(7 HEART)(10 DIAMOND)(J CLUB)(2 CLUB)
(7 HEART)(10 DIAMOND)(2 CLUB)(A SPADE)(J CLUB)
(2 CLUB)(7 HEART)(10 DIAMOND)(A SPADE)(J CLUB)
```

Изглежда програмата работи коректно – всеки път извежда различна подредба на картите. Пускаме още няколко примера и виждаме, че работи правилно и за тях. Готови сме.

Стъпка 5 – Вход от конзолата

Остава да реализираме вход от конзолата, за да дадем възможност на потребителя да въведе картите, които да бъдат разбъркани. Забележете, че оставихме за накрая тази стъпка. Защо? Ами много просто. Нали не искаме всеки път при стартиране на програмата да въвеждаме 5 карти само за да тестваме дали някаква малка част от кода работи коректно (преди цялата програма да е написана докрай)? Като кодираме твърдо входните данни си спестяваме много време за въвеждането им по време на разработка.

Ако задачата изисква вход от конзолата, реализирайте го най-накрая, след като всичко останало работи. Докато

	пишете програмата, тествайте с твърдо кодирани примерни данни, за да не въвеждате входа всеки път. Така ще спестите много време и нерви.

Въвеждането на входните данни е хамалска задача, която всеки може да реализира. Трябва само да се помисли в какъв формат се въвеждат картите и дали се въвеждат една по една или всички на един път и дали лицето и боята се задават наведнъж или поотделно, в това няма нищо сложно. Нека оставим тази част за упражнение. Вече изложихме принципите при имплементацията на кода, а самият код е лесно да се напише.

Сортиране на числа – стъпка по стъпка

До момента ви показахме колко важно е да пишете програмата си стъпка по стъпка и преди да преминете на следващата стъпка да се убедите, че предходната е реализирана качествено и работи коректно.

За задачата със сортиране на числа в нарастващ ред нещата не стоят по-различно. Отново правилният подход към имплементацията изисква да работим на стъпки. Нека видим накратко кои са стъпките. Няма да пишем кода, но ще набележим основните моменти, през които трябва да преминете. Да предположим, че реализираме идеята за сортиране чрез **`ArrayList`**, в който последователно намираме най-малкото число, отпечатваме го и го изтриваме. Ето какви биха могли да са стъпките:

Стъпка 1. Измисляме подходящ пример, с който ще си тестваме. Създаваме **`ArrayList<Integer>`** и го запълваме с числата от нашия пример. Реализираме отпечатване на числата.

Стартираме програмата и тестваме.

Стъпка 2. Реализираме метод, който намира най-малкото число в масива и връща позицията му.

Тестваме метода за търсене на най-малко число. Пробваме различни поредици числа, за да се убедим, че търсенето работи коректно (слагаме най-малкия елемент в началото, в края, в средата; пробваме и когато най-малкия елемент се повтаря няколко пъти).

Стъпка 3. Реализираме метод, който намира най-малкото число, отпечатва го и го изтрива.

Тестваме с нашия пример дали методът работи коректно.

Стъпка 4. Реализираме метода, който сортира числата. Той изпълнява предходния метод N пъти (където е броят на числата).

Задължително тестваме дали всичко работи както трябва.

Стъпка 5. Ако е необходим вход от конзолата, реализираме го.

Виждате, че подходът с разбиването на стъпки е приложим при всякакви задачи. Просто трябва да съобразим кои са нашите елементарни стъпки при имплементацията и да ги изпълняваме една след друга, като не забравяме да тестваме всяко парче код възможно най-рано. След всяка стъпка е хубаво да стартираме програмата, за да се убедим, че до този момент всичко работи правилно. Така ще откриваме евентуални проблеми още при възникването им и ще ги отстраняваме бързо и лесно.

Тествайте решението си!

"Аз съм готов с първа задача. Веднага трябва да започна следващата." На всеки му е хрумвала такава мисъл, когато бил е на изпит. В програмирането, обаче, тази мисъл означава следното:

1. Аз съм разбрал добре условието на задачата.
2. Аз съм измислил алгоритъм за решаването на задачата.
3. Аз съм тествал на лист хартия моя алгоритъм и съм се уверил, че е правилен.
4. Аз съм помислил за структурите от данни и за ефективността на моя алгоритъм.
5. Аз съм написал програма, която реализира коректно моя алгоритъм.
6. Аз съм тествал обстойно моята програма с подходящи примери, за да се уверя, че работи коректно, дори в необичайни ситуации.

Неопитните програмисти почти винаги пропускат последната точка. Те смятат, че тестването не е тяхна задача, което е най-голямата им грешка. Все една да смятаме, че Майкрософт не са длъжни да тестват Windows и могат да оставят той да "гърми" при всяко второ натискане на мишката.

Тестването е неразделна част от програмирането! Да пишеш код, без да го тестваш е като да пишеш на клавиатурата без виждаш екрана на компютъра – мислиш си, че пишеш правилно, но най-вероятно имаш грешки.

Опитните програмисти знаят, че ако напишат код и той не е тестван, това означава, че той още не е завършен. В повечето софтуерни фирми е недопустимо да се предаде код, който не е тестван. В софтуерната индустрия дори е възприета концепцията за "unit testing" – автоматизирано тестване на отделните единици от кода (методи, класове и цели модули). Unit testing означава да пишем програма, която тества нашата програма дали работи коректно. В някои фирми дори първо се измислят тестовите сценарии, пишат се тестовете за програмата и най-накрая се пише самата програма. Темата за unit testing е много сериозна и обемна, но с нея ще се запознаете по-късно, когато навлезете в дълбините на професията "софтуерен инженер". Засега, нека се фокусираме върху ръчното тестване,

което всеки един програмист може да извърши, за да се убеди, че неговата програма работи коректно.

Как да тестваме?

Една програма е коректна, ако работи коректно за всеки валиден набор от входни данни. Тестването е процес, който цели да установи наличие на дефекти в програмата, ако има такива. То не може да установи със сигурност дали една програма е коректна, но може да провери в голяма степен дали в програмата има дефекти, които причиняват некоректни резултати или други проблеми.

За съжаление всички възможни набори входни данни за една програма обикновено са неизброимо много и не може да се тества всеки от тях. Затова в практиката на софтуерното тестване се подготвят и изпълняват такива набори от входни данни (тестове), които целят да обхванат максимално пълно всички различни ситуации (случаи на употреба), които възникват при изпълнение на програмата. Този набор има за цел с минимални усилия (т. е. с минимален брой и максимална простота на тестовете) да провери всички основни случаи на употреба. Ако при тестването по този начин не бъдат открити дефекти, това не доказва, че програмата е 100% коректна, но намалява в голяма степен вероятността на по-късен етап да се наблюдават дефекти и други проблеми.

Тестването може да установи само наличие на дефекти. То не може да докаже, че дадена програма е коректна! Програмите, които са тествани добре имат много по-малко дефекти, отколкото програмите, които изобщо не са тествани или не са тествани качествено.

Тестването е добре да започва от един пример, с който обхващаме типичния случай в нашата задача. Той най-често е същият пример, който сме тествали на хартия и за който очакваме нашият алгоритъм да работи коректно. След написване на кода обикновено следва отстраняване на поредица от дребни грешки и най-накрая нашият пример тръгва. След това е нормално да тестваме програмата с по-голям и по-сложен пример, за да видим как се държи тя в по-сложни ситуации. Следва тестване на граничните случаи и тестване за бързодействие. В зависимост от сложността на конкретната задача могат да се изпълнят от един-два до няколко десетки теста, за да се покрият всички основни случаи на употреба.

Тестване с добър представител на общия случай

Както вече споменахме, нормално е тестването да започне с тестов пример, който е добър представител на общия случай. Това е тест, който хем е достатъчно прост, за да бъде проигран ръчно на хартия, хем е достатъчно общ, за да покрие общия случай на употреба на програмата, а не някой

частен случай. Следвайки този подход най-естественото нещо, което някой програмист може да направи е следното:

1. Да измисли пример, който е добър представител на общия случай.
2. Да тества примера на ръка (на хартия).
3. Да очаква примера да тръгне успешно и от имплементацията на неговия алгоритъм.
4. Да се убеди, че примерът му работи коректно след написване на програмата и отстраняване на дребните грешки, които възникват при писането на кода.

За съжаление много програмисти спират с тестването в този момент. Някои по-неопитни програмисти правят дори нещо по-лошо: измислят какъв да е пример (който е прост частен случай на задачата), не го тестват на хартия, пишат някакъв код и накрая като тръгне този пример, решават, че са приключили. Не правете така! Това е като да ремонтираш лека кола и когато си готов, без да запалиш двигателя да пуснеш колата по някой наклон и като тръгне надолу да кажеш "Готова е колата. Ето, движи се без никакъв проблем."

Какво още да тестваме?

Тестването на примера, който сте проиграли на хартия е едва първата стъпка от тестването на програмата. Следва да извършите още няколко задължителни теста, с които да се убедите, че програмата ви работи коректно:

- Сериозен тест за обичайния случай. Целта на този тест е да провери дали за по-голям и по-сложен пример вашата програма работи коректно. За нашата задача с разбъркването на картите такъв тест може да е тесте от 52 карти.
- Тестове за граничните случаи. Те проверяват дали вашата програма работи коректно при необичаен вход на границата на допустимото. За нашата задача такъв пример е разбъркването на тесте, което се състои само от една карта.
- Тестове за бързодействие. Тези тестове поставят програмата в екстремални условия като ѝ подават големи по размерност входни данни и проверяват бързодействието.

Нека разгледаме горните групи тестове една по една.

Сериозен тест на обичайния случай

Вече сме тествали програмата за един случай, който сме измислили на ръка и сме проиграли на хартия. Тя работи коректно. Този случай покрива типичния сценарий за употреба на програмата. Какво повече трябва да

тестваме? Ами много просто, възможно е програмата да е грешна, но да работи по случайност за нашия случай.

Как да подготвим по-сериозен тест? Това зависи много от самата задача. Тестът хем трябва да е с по-голям обем данни, отколкото ръчния тест, но все пак трябва да можем да проверим изхода от програмата дали е коректен.

За нашия пример с разбъркването на карти в случаен ред е нормално да тестваме с пълно тесте от 52 карти. Лесно можем да произведем такъв входен тест с два вложени цикъла. След изпълнение на програмата лесно можем да проверим дали резултатът е коректен – трябва картите да са разбъркани и разбъркването да е случайно. Необходимо е още при две последователни изпълнения на този тест да се получи тотално различно разбъркване. Ето как изглежда кода, реализиращ такъв тест:

Test52Cards.java

```
import java.util.ArrayList;

public class Test52Cards {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      String[] allFaces = new String[] {"2", "3", "4", "5",
         "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
      Suit[] allSuits = new Suit[] {
         Suit.CLUB, Suit.DIAMOND, Suit.HEART, Suit.SPADE};
      for (String face : allFaces) {
         for (Suit suit : allSuits) {
            Card card = new Card(face, suit);
            cards.add(card);
         }
      }

      CardsShuffle.shuffleCards(cards);
      CardsShuffle.printCards(cards);
   }
}
```

Ако го изпълним няколко пъти подред получаваме примерно такъв резултат:

```
(J HEART)(10 CLUB)(4 HEART)(2 SPADE)(3 HEART)(3 DIAMOND)(2 HEART)(3
SPADE)(4 CLUB)(4 DIAMOND)(6 CLUB)(J SPADE)(5 CLUB)(5 DIAMOND)(A
SPADE)(K SPADE)(4 SPADE)(6 DIAMOND)(A DIAMOND)(6 SPADE)(7 CLUB)(10
SPADE)(9 DIAMOND)(A HEART)(Q SPADE)(8 DIAMOND)(8 HEART)(8 SPADE)(9
CLUB)(Q DIAMOND)(9 HEART)(9 SPADE)(Q HEART)(10 DIAMOND)(2 DIAMOND)(6
HEART)(J CLUB)(J DIAMOND)(Q CLUB)(7 DIAMOND)(5 SPADE)(2 CLUB)(5
```

```
HEART)(10 HEART)(K CLUB)(3 CLUB)(K HEART)(8 CLUB)(A CLUB)(K
DIAMOND)(7 HEART)(7 SPADE)
...
(Q HEART)(3 CLUB)(2 HEART)(2 DIAMOND)(9 SPADE)(3 DIAMOND)(3 HEART)(3
SPADE)(7 CLUB)(9 CLUB)(9 DIAMOND)(4 SPADE)(5 CLUB)(5 DIAMOND)(10
SPADE)(5 SPADE)(J HEART)(6 DIAMOND)(Q DIAMOND)(4 HEART)(5 HEART)(7
HEART)(J CLUB)(4 CLUB)(2 SPADE)(K SPADE)(8 HEART)(Q CLUB)(6 CLUB)(6
HEART)(9 HEART)(A SPADE)(J SPADE)(J DIAMOND)(10 HEART)(10 DIAMOND)(K
CLUB)(K HEART)(8 CLUB)(4 DIAMOND)(Q SPADE)(6 SPADE)(A DIAMOND)(10
CLUB)(8 DIAMOND)(7 SPADE)(K DIAMOND)(2 CLUB)(A CLUB)(7 DIAMOND)(A
HEART)(8 SPADE)
...
```

Изглежда, че картите са подредени случайно и са различни при всяко изпълнение на програмата. Няма видими дефекти (примерно повтарящи се или липсващи карти). Програмата работи бързо и не зависва. Изглежда сме се справили добре.

Нека вземем друга задача: сортиране на числа. Как да си направим сериозен тест за обичайния случай? Ами най-лесното е да генерираме поредица от 100 или дори 1000 случайни числа и да ги сортираме. Проверката за коректност е лесна: трябва числата да са подредени по големина. Друг тест, който е удачен при сортирането на числа е да вземем числата от 1000 до 1 в намаляващ ред и да ги сортираме. Трябва да получим същите числа, но сортирани в нарастващ ред от 1 до 1000. Би могло да се каже, че това е най-трудния възможен тест за тази задача и ако той работи за голям брой числа, значи програмата се очаква да работи добре.

Нека разгледаме и другите тестове, които е добре винаги да правим.

Гранични случаи

Най-честото нещо, което се пропуска при решаването на задачи, пък и въобще в програмирането, е да се помисли за граничните ситуации. Граничните ситуации се получават при входни данни на границата на нормалното и допустимото. При тях често пъти програмата гърми, защото не очаква толкова малки или големи или необичайни данни, но те все пак са допустими по условие или не са допустими, но не са предвидени.

Как да тестваме граничните ситуации? Ами разглеждаме всички входни данни, които програмата получава и се замисляме какви са екстремните им стойности и дали са допустими. Възможно е да имаме екстремно малки стойности, екстремно големи стойности или просто странни комбинации от стойности. Ако по условие имаме ограничения, примерно до 52 карти, стойностите около това число 52 също са гранични и могат да причинят проблеми.

Граничен случай: разбъркване на една карта

Например в нашата задача за разбъркване на карти граничен случай е да разбъркаме една карта. Това е съвсем валидна ситуация (макар и необичайна), но нашата програма би могла да не работи коректно за една карта нея поради някакви особености. Нека проверим какво става при разбъркване на една карта. Можем да напишем следния малък тест:

ShuffleOneCard.java

```
import java.util.ArrayList;

public class ShuffleOneCard {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      cards.add(new Card("A", Suit.CLUB));
      CardsShuffle.shuffleCards(cards);
      CardsShuffle.printCards(cards);
   }
}
```

Изпълняваме го и получаваме напълно неочакван резултат:

```
Exception in thread "main" java.lang.IllegalArgumentException: n must
be positive
   at java.util.Random.nextInt(Random.java:250)
   at CardsShuffle.performSingleExchange(CardsShuffle.java:24)
   at CardsShuffle.shuffleCards(CardsShuffle.java:33)
   at ShuffleOneCard.main(ShuffleOneCard.java:7)
```

Ясно е какъв е проблемът: генерирането на случайно число се счупи, защото му се подава отрицателен диапазон. Нашата програма работи добре при нормален брой карти, но не работи за една карта. Открихме лесен за отстраняване дефект, който бихме пропуснали с лека ръка, ако се бяхме разгледали внимателно граничните случаи. След като знаем какъв е проблемът поправката на кода е тривиална:

```
public static void shuffleCards(ArrayList<Card> cards) {
   if (cards.size() > 1) {
      for (int i=1; i<=cards.size(); i++) {
         performSingleExchange(cards);
      }
   }
}
```

Тестваме отново и се убеждаваме, че проблемът е решен.

Граничен случай: разбъркване на две карти

Щом има проблем за 1 карта, сигурно може да има проблем и за 2 карти. Не звучи ли логично? Нищо не ни пречи да проверим. Стартираме програмата с 2 карти няколко пъти очакваме да получим различни разместванията на двете карти. Ето примерен код, с който можем да направим това:

ShuffleTwoCards.java

```
import java.util.ArrayList;

public class ShuffleOneCard {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      cards.add(new Card("A", Suit.CLUB));
      cards.add(new Card("3", Suit.CLUB));
      CardsShuffle.shuffleCards(cards);
      CardsShuffle.printCards(cards);
   }
}
```

Стартираме няколко пъти и резултатът е все един и същ:

```
(A CLUB)(3 CLUB)
```

Изглежда пак нещо не е наред. Ако разгледаме кода или го пуснем през дебъгера, ще се убедим, че всеки път се прави точно едно разместване на първата карта с втората и при две карти няма как да се получи случайно разместване. Как да решим проблема? Веднага можем да се сетим за няколко решения:

- Правим единичното разместване N+K брой пъти, където K е случайно число между 0 и 1.
- При разместванията допускаме случайната позиция, на която отива първата карта да включва и нулевата позиция.
- Разглеждаме случая с 2 карти като специален и пишем отделен метод специално за него.

Първото решение изглежда най-просто за имплементация. Да го пробваме. Получаваме следния код:

```
public static void shuffleCards(ArrayList<Card> cards) {
   if (cards.size() > 1) {
      Random rand = new Random();
      int exchangesCount = cards.size() + rand.nextInt(2);
      for (int i=1; i<=exchangesCount; i++) {
```

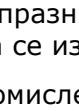
```
            performSingleExchange(cards);
        }
    }
}
```

Тестваме отново разбъркването на две карти и този път изглежда, че програмата работи коректно.

Щом има проблем за 2 карти, може да има проблем и за 3 карти, нали? Ако тестваме програмата за 3 карти, ще се убедим, че тя работи коректно. След няколко стартирания получаваме всички възможни разбърквания на трите карти, което показва, че случайното разбъркване може да получи всички пермутации на трите карти. Този път не открихме дефекти и програмата няма нужда от промяна.

Граничен случай: разбъркване на нула карти

Какво още може да проверим? Има ли други необичайни, гранични ситуации. Да помислим. Какво ще стане, ако се опитаме да разбъркаме празен списък от карти? Това наистина е малко странно, но има едно правило, че една програма трябва или да работи коректно или да сигнализира за грешка. Нека да видим какво ще върне нашата програма за 0 карти. Резултатът е празен списък. Коректен ли е? Ами да, ако разбъркаме 0 карти в случаен ред би трябвало да получим пак 0 карти. Изглежда всичко е наред.



При грешни входни данни програмата не трябва да връща грешен резултат, а трябва или да върне верен резултат или да съобщи, че входните данни са грешни.

Какво мислите за горното правило? Логично е нали? Представете си, че правите програма, която показва графични изображения (снимки). Какво става при снимка, която представлява празен файл. Това е също необичайна ситуация, която не би трябвало да се случва, но може да се случи. Ако при празен файл вашата програма зависва или хвърля необработено изключение, това би било много досадно за потребителя. Нормално е празният файл да бъде изобразен със специална икона или вместо него да се изведе съобщение "Invalid image file", нали?

Помислете колко гранични и необичайни ситуации има в Windows. Какво става ако печатаме празен файл на принтера? Дали Windows забива в този момент и показва небезизвестния "син екран"? Какво става, ако в калкулатора на Windows направим деление на нула? Какво става, ако копираме празен файл (с дължина 0 байта) с Windows explorer? Какво става, ако в Notepad се опитаме да създадем файл без име (с празен стринг, зададен като име)? Виждате, че гранични ситуации има много и навсякъде. Наша задача като програмисти е да ги улавяме и да мислим за тях преди още да се случат, а не едва когато неприятно развълнуван потребител

яростно ни нападне по телефона с неприлични думи по адрес на наши близки роднини.

Да се върнем на нашата задача за разбъркване на картите. Оглеждайки се за гранични и необичайни случаи се сещаме дали можем да разбъркаме -1 карти? Понеже няма как да създадем масив с -1 елемента, считаме, че такъв случай няма как да се получи.

Понеже нямаме горна граница на картите, няма друга специална точка (подобна на ситуацията с 1 карта), около която да търсим за специални ситуации. Прекратяваме търсенето на гранични случаи около броя на картите. Изглежда предвидихме всички ситуации.

Остава да се огледаме дали няма други стойности от входните данни, които могат да причинят проблеми, примерно невалидна карта, карта с невалидна боя, карта с отрицателно лице (примерно -1 спатия) и т.н. като се замислим нашия алгоритъм не се интересува какво точно разбърква (карти за игра или яйца за омлет), така че това не би трябвало да е проблем. Ако имаме съмнения, можем на си направим тест и да се убедим, че при невалидни карти резултатът от разбъркването им не е грешен.

Оглеждаме се за други гранични ситуации във входните данни и не се сещаме за такива. Остава единствено да измерим бързодействието, нали? Всъщност пропуснахме нещо много важно: да тестваме всичко наново след поправките.

Повторно тестване след корекциите (regression testing)

Често пъти при корекции на грешки се получават нови грешки, които преди не са съществували. Например, ако поправим грешката за 2 карти чрез промяна на правилата за размяна на единична карта, това би могло да доведе до грешен резултат при или повече 3 карти. При всяка промяна, която би могла да засегне други случаи на употреба, е редно да пускаме отново тестовете, които сме правили до момента, за да сме сигурни, че промяната не повреждa вече работещите случаи. За тази цел е добре да запазваме тестовете на програмата, които сме изпълнявали, а не да ги изтриваме.

Идеята за повторяемост на тестовете лежи в основата на концепцията unit testing, но тази тема, както вече споменахме е за по-напреднали и затова я оставаме за по-нататък във времето (и пространството).

Когато сте открили и сте поправили грешка в кода, отнасяща се за някой специфичен тест, уверете се, че поправката не засяга всички останали тестове. За целта е препоръчително да запазвате всички тестове, които изпълнявате.

Тестове за производителност

Нормално е винаги, когато пишете софтуер, да имате някакви изисквания и критерии за бързодействие на програмите или модулите, които пишете. Никой не обича машината му да работи бавно, нали? Затова трябва да се стремите да не пишете софтуер, който работи бавно, освен, ако нямате добре причина за това.

Как тестваме бързодействието (производителността) на програмата. Първият въпрос, който трябва да си зададем, когато стигнем до тестване на бързодействието, е имаме ли изисквания за скорост. Ако имаме какви са те? Ако нямаме какви ориентировъчни критерии за бързодействие трябва да спазим?

Разбъркване на карти – тестове за производителност

Нека да разгледаме за пример нашата програма за разбъркване на тесте карти. Какви изисквания за бързодействие би могла да има тя? Първо имаме ли по услови такива изисквания? Нямаме изрично изискване в стил "програмата трябва да завършва за една секунда или по-бързо при 500 карти на съвременна компютърна конфигурация". Щом нямаме такива изрични изисквания, все пак трябва някак да решим въпроса с оценката на бързодействието, неформално, по усет.

Понеже работим с карти за игра, считаме, че едно тесте има 52 карти. Вече пускахме такъв тест и видяхме, че работи мигновено, т.е. няма видимо забавяне. Изглежда за нормалния случай на употреба бързината не създава проблеми.

Нормално е да тестваме програмата и с много повече карти, примерно с 52 000, защото в някой специален случай някой може да реши да разбърква много карти и да има проблем. Лесно можем да си направим такъв пример като добавим 1000 пъти нашите 52 карти и ги разбъркаме. Нека пуснем един такъв пример:

Shuffle52000Cards.java

```
import java.util.ArrayList;

public class Test52000Cards {
   public static void main(String[] args) {
      ArrayList<Card> cards = new ArrayList<Card>();
      String[] allFaces = new String[] {"2", "3", "4", "5",
           "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
         Suit[] allSuits = new Suit[] {
            Suit.CLUB, Suit.DIAMOND, Suit.HEART, Suit.SPADE};
      for (int count = 1; count<=1000; count++) {
         for (String face : allFaces) {
            for (Suit suit : allSuits) {
```

```
            Card card = new Card(face, suit);
            cards.add(card);
         }
      }
   }

   CardsShuffle.shuffleCards(cards);
   CardsShuffle.printCards(cards);
  }
}
```

Стартираме програмата и забелязваме, че машината леко се успива за около десетина секунди. Разбира се при по-бавни машини успиването е за по-дълго. Какво се случва? Би трябвало при 52 000 карти да направим приблизително толкова единични разместванията, а това би трябвало да отнеме частица от секундата. Защо имаме секунди забавяне? Опитните програмисти веднага ще се сетят, че печатаме големи обеми информация на конзолата, а това е бавна операция. Ако коментираме реда, в който отпечатваме резултата и измерим времето за изпълнение на разбъркването на картите, ще се убедим, че програмата работи достатъчно бързо дори и за 52 000 карти. Ето как можем да замерим времето:

Shuffle52000Cards.java

```
import java.util.ArrayList;

public class Test52000Cards {
  public static void main(String[] args) {
    ...
    long oldTime = System.currentTimeMillis();
    CardsShuffle.shuffleCards(cards);
    long newTime = System.currentTimeMillis();
    System.out.printf("Execution time: %d ms",
      newTime-oldTime);
    //CardsShuffle.printCards(cards);
  }
}
```

Можем да видим точно колко време отнема изпълнението на метода за разбъркване на картите:

```
Execution time: 31 ms
```

Изглежда напълно приемливо. Нямаме проблем с бързодействието.

Сортиране на числа – тестове за производителност

Нека разгледаме друга задача: сортиране на масив с числа. При нея бързодействието може да се окаже много по-проблемно, отколкото разбъркването на тесте карти. Нека сме направили просто решение, което работи така: намира най-малкото число в масива и го разменя с числото на позиция 0. След това намира сред останалите числа най-малкото и го поставя на позиция 1. Това се повтаря докато се стигне до последното число, което би трябвало да си е вече на мястото. Няма да коментираме верността на този алгоритъм. Той е добре известен под името "метод на пряката селекция".

Сега да предположим, че сме минали през всички стъпки за решаването на задачи по програмиране и накрая сме стигнали до този пример, с който се опитваме да сортираме 10 000 случайни числа:

Sort10000Numbers.java

```
import java.util.Arrays;
import java.util.Random;

public class SortNumbers {
   public static void main(String[] args) {
      int[] numbers = new int[10000];
      Random rnd = new Random();
      for (int i=0; i<numbers.length; i++) {
         numbers[i] = rnd.nextInt(2 * numbers.length);
      }

      sortNumbers(numbers);

      System.out.println(Arrays.toString(numbers));
   }

   private static void sortNumbers(int[] numbers) {
      for (int i=0; i<numbers.length-1; i++) {
         int minIndex = i;
         for (int j=i+1; j<numbers.length; j++) {
            if (numbers[j] < numbers[minIndex]) {
               minIndex = j;
            }
         }
         int oldNumber = numbers[i];
         numbers[i] = numbers[minIndex];
         numbers[minIndex] = oldNumber;
      }

   }
```

```
}
```

Стартираме го и изглежда, че той работи за под секунда на нормална съвременна машина. Резултатът (със съкращения) би могъл да е нещо такова:

```
[0, 14, 19, 20, 20, 22, ..., 19990, 19993, 19995, 19996]
```

Сега правим още един експеримент за 300 000 случайни числа и виждаме, че програмата като че ли зависва или работи прекалено бавно, за да я изчакаме. Това е сериозен проблем с бързодействието.

Преди да се втурнем да го решаваме трябва, обаче, да си зададем един много важен въпрос: дали ще имаме реална ситуация, при която ще се наложи да сортираме 300 000 числа. Ако сортираме примерно оценките на студентите в един курс, те не могат да бъдат повече от няколко десетки. Ако, обаче, сортираме цените на акциите на голяма софтуерна компания за цялата й история на съществуване на фондовата борса, можем да имаме огромен брой числа, защото цената на акциите й може да се променя всяка секунда. За десетина години цените на акциите на тази компания биха могли да се променят няколкостотин милиона пъти. В такъв случай трябва да търсим по-ефективен алгоритъм за сортиране.

Как да правим ефективно сортиране на цели числа можем да прочетем в десетки сайтове в Интернет и в класическите книги по алгоритми. Конкретно за тази задача подходящо е да използваме алгоритъма за сортиране "radix sort" (http://en.wikipedia.org/wiki/Radix_sort), но тази дискусия е извън темата и ще я пропуснем.

Нека припомним доброто старо правило за ефективността:

Винаги трябва да правим компромис между времето, за което ще напишем програмата и бързодействието, което искаме да постигнем. Иначе може да изгубим време да решаваме проблем, който не съществува или да дадем решение, което не върши работа.

Трябва да имаме предвид и че за някои задачи изобщо не съществуват бързи алгоритми и ще трябва да се примирим с ниската производителност. Например за задачата за намиране на всички прости делители на цяло число (вж. http://en.wikipedia.org/wiki/Integer_factorization) няма известно бързо решение.

За някои задачи нямаме нужда от бързина, защото очакваме входните данни да са достатъчно малки и тогава е безумно да търсим сложни алгоритми с цел бързодействие. Например задачата за сортиране на оценките на студентите от даден курс може да се реши с произволен

алгоритъм за сортиране и при всички случаи ще работи бързо, тъй като броят на студентите се очаква да е достатъчно малък.

Генерални изводи

Преди да започнете да четете настоящата тема сигурно сте си мислили, че това ще е най-скучната и безсмислена до момента, но вярвам, че сега мислите по съвсем различен начин. Всички си мислят, че знаят как да решават задачи по програмиране и че за това няма "рецепта" (просто трябва да го можеш), но въобще не е така. Има си рецепта и ние ви я показахме в действие!

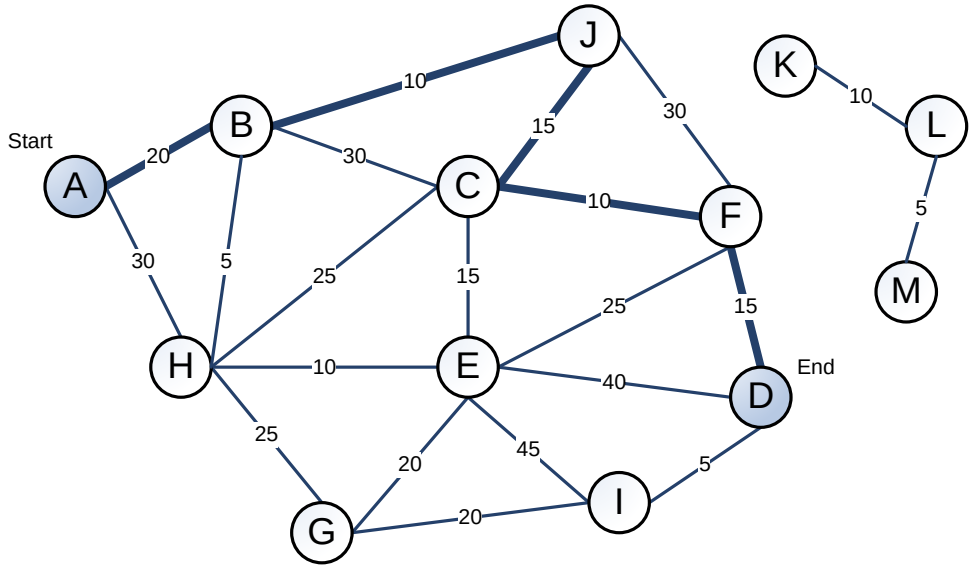
Само се замислете колко грешки и проблеми открихме докато решавахме една супер лесна и проста задача: разбъркване на карти. Щяхме ли да напишем качествено решение, ако не бяхме подходили към задачата по рецептата, изложена по-горе? А какво би се случило, ако решаваме някоя много по-сложна и трудна задача, примерно да намерим оптимален път през сутрешните задръствания в София по карта на града с актуални данни за трафика? При такива задачи е абсолютно немислимо да подходим хазартно и да се хвърлим на първата идея, която ни дойде на ум. Първата стъпка към придобиване на умения за решаване на такива сложни задачи е да се научите да подхождате към задачата систематично и да усвоите рецептата за решаване на задачи, която ви демонстрирахме в действие. Това, разбира се съвсем няма да ви е достатъчно, но е силна крачка напред!

За решаването на задачи по програмиране си има рецепта! Ползвайте систематичен подход и ще имате много по-голям успех, отколкото, ако карате по усет. Дори професионалистите с десетки години опит ползват в голяма степен описания от нас подход. Ползвайте го и вие и ще се убедите, че помага!

Упражнения

1. Използвайки описаната в тази глава методология за решаване на задачи по програмиране решете следната задача: разполагаме с карта на един град. Картата се състои от улици и кръстовища. За всяка улица на картата е отбелязана нейната дължината. Едно кръстовище свързва няколко улици. Задачата е да се намери и отпечата най-късият път между двойка кръстовища (измерен като суми от дължините на улиците, през които се преминава).

   Ето как изглежда схематично картата на един примерен град:



На тази карта най-късият път между кръстовища A и D е с дължина 70 и е показан на фигурата с удебелени линии. Както виждате, между A и D има много пътища с най-различна дължина. Не винаги най-късото начало води към най-късия път и не винаги най-малкият брой улици води до най-къс път. Между някои двойки кръстовища дори въобще не съществува път. Това прави задачата доста интересна.

Входните данни се задават в текстов файл **`map.txt`**. Файлът започва със списък от улици и техните дължини, след което следва празен ред и след него следват двойки кръстовища, между които се търси най-краткия път. Файлът завършва с празен ред:

```
A B 20
A H 30
B H 5
...
L M 5
(празен ред)
A D
H K
A E
(празен ред)
```

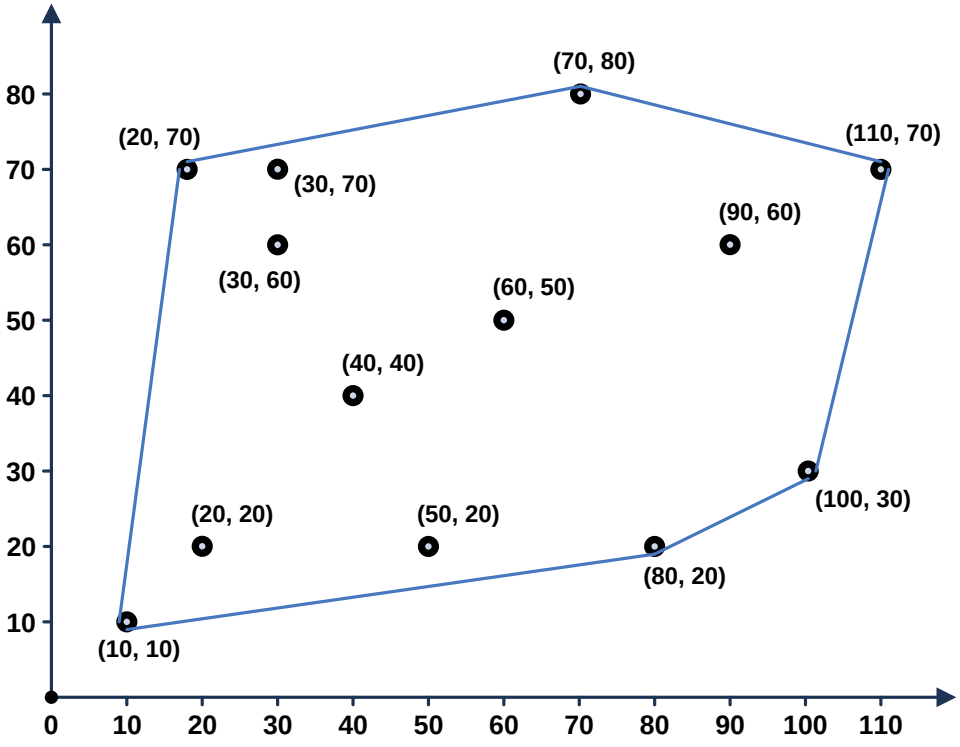
Резултатът от изпълнението на програмата за всяка двойка кръстовища от списъка във входния файл трябва да е дължината на най-късия път, следвана от самия път. За картата от нашия пример изходът трябва да изглежда така:

```
70 ABJCFD
```

```
No path!
35 ABHE
```

2. * В равнината са дадени са N точки с координати цели, положителни числа. Тези точки представляват дръвчета в една нива. Стопанинът на нивата иска да огради дръвчетата, като използва минимално количество ограда. Напишете програма, която намира през кои точки трябва да минава оградата. Използвайте методологията за решаване на задачи по програмиране!

   Ето как би могла да изглежда градината:



   Входните данни се четат от файл **garden.txt**. На първия ред на файла е зададен броя точки. Следват координатите на точките. За нашия пример входният файл би могъл да има следното съдържание:

```
13
60 50
100 30
40 40
20 70
50 20
30 70
10 10
```

```
110 70
90 60
80 20
70 80
20 20
30 60
```

Изходните данни трябва да се отпечатат на конзолата в като последователност от точки, през които оградата трябва да мине. Ето примерен изход:

```
(10, 10) - (20, 70) - (70, 80) - (110, 70) - (100, 30) - (80, 20)
- (10, 10)
```

Решения и упътвания

1. Следвайте стриктно методологията за решаване на задачи по програмиране! Задачата е сложна и изисква да й отделите повече внимание. Първо си нарисувайте примера на хартия. Опитайте се да измислите сами правилен алгоритъм за намиране на най-къс път. След това потърсете в Интернет по ключови думи "shortest path algorithm". Много е вероятно бързо да намерите статия с описание на алгоритъм за най-къс път.

   Проверете дали алгоритъмът е верен. Пробвайте различни примери.

   В каква структура от данни ще пазите картата на града? Помислете кои са операциите, които ви трябват в алгоритъма за най-къс път. Вероятно ще стигнете до идеята да пазите списък от улиците за всяко кръстовище, а кръстовищата да пазите в списък или хеш-таблица.

   Помислете за ефективността. Ще работи ли вашият алгоритъм за 1 000 кръстовища и 5 000 улици?

   Пишете стъпка по стъпка. Първо направете четенето на входните данни. Реализирайте отпечатване на прочетените данни. Реализирайте алгоритъма за най-къс път. Ако можете, разбийте реализацията на стъпки. Например като за начало можете да търсите само дължината на най-късия път без самия път (като списък от кръстовища), защото е полесно. Реализирайте след това и намирането на самия най-къс път. Помислете какво става, ако има няколко най-къси пътя с еднаква дължина. Накрая реализирайте изхода, както се изисква в условието на задачата.

   Тествайте решението си! Пробвайте с празна карта. Пробвайте с карта с 1 кръстовище. Пробвайте случай, в който няма път между зададените кръстовища. Пробвайте с голяма карта (1 000 кръстовища и 5 000 улици). Можете да си генерирате такава с няколко реда програмка. За имената на кръстовищата трябва да използвате **`String`**, а не **`char`**, нали?

Иначе как ще имате 1 000 кръстовища? Работи ли бързо? Работи ли вярно?

Внимавайте с входните и изходните данни. Спазвайте форматът, който е указан в условието на задачата!

2. Ако не сте много силни в аналитичната геометрия, едва ли ще измислите решение на задачата сами. Опитайте търсене в Интернет по ключовите думи "convex hull algorithm". Знаейки, че оградата, която трябва да построим се нарича " изпъкнала обвивка" (convex hull) на множество точки в равнината, ще намерим стотици статии в Интернет по темата, в някои, от които дори има сорс код на Java. Не преписвайте грешките на другите и особено сорс кода! Мислете! Проучете как работи алгоритъма и си го реализирайте сами.

   Проверете дали алгоритъмът е верен. Пробвайте различни примери. Какво става, ако има няколко точки на една линия върху изпъкналата обвивка? Трябва ли да включвате всяка от тях? Помислете какво става, ако има няколко изпъкнали обвивки. От коя точка започвате? По часовниковата стрелка ли се движите или обратното? В условието на задачата има ли изискване как точно да са подредени точките в резултата?

   В каква структура от данни ще пазите точките? В каква структура ще пазите изпъкналата обвивка?

   Помислете за ефективността. Ще работи ли алгоритъмът за 1 000 точки?

   Пишете стъпка по стъпка. Първо направете четенето на входните данни. Реализирайте отпечатване на прочетените точки. Реализирайте алгоритъма за изпъкнала обвивка. Ако можете, разбийте реализацията на стъпки. Накрая реализирайте изхода, както се изисква в условието на задачата.

   Тествайте решението си! Какво става, ако имаме 0 точки? Пробвайте с 1 точка. Пробвайте с 2 точки. Пробвайте с 5 точки, които са на една линия. Работи ли алгоритъмът ви? Какво става при 10 точки и още 10, които съвпадат с първите 10? Какво става, ако имаме 10 точки, всичките една върху друга? Какво става, ако имаме много точки, примерно 1 000. Работи ли бързо вашият алгоритъм? Какво става, ако координатите на точките са големи числа, примерно (100 000 000, 200 000 000)? Влияе ли това на алгоритъма? Има ли грешки от загуба на точност?

   Внимавайте с входните и изходните данни. Спазвайте формата, който е указан в условието на задачата! Не си измисляйте сами формата на входа и изхода. Те са дефинирани по условие.

   Ако имате мерак, направете си визуализация на точките и изпъкналата обвивка. Направете си и генератор на случайни тестови данни и си тествайте многократно решението, като гледате визуализацията на обвивката – дали коректно обвива точките и дали е минимална.

Глава 23. Примерен изпит по програмиране – 30.09.2005 г.

Автори

Стефан Стаев

Светлин Наков

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на три примерни задачи от изпит по програмиране в НАРС, проведен на 30.09.2005 г. При решаването им ще приложим на практика описаната методология в главата "Как да решаваме задачи по програмиране".

Задача 1: Извличане на текста от HTML документ

Даден е HTML файл с име **`Problem1.html`**. Да се напише програма, която отстранява от него всички HTML тагове и запазва само текста вътре в тях. Изходът да се изведе във файла **`Problem1.txt`**.

Примерен входен файл **`Problem1.html`**:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Примерен изходен файл **`Problem1.txt`**:

```
Welcome to our site!
Home
Contacts
About
```

Измисляне на идея за решение

Първото, което ни хрумва, като идея за решение на тази задача е да четем последователно (примерно ред по ред или буква по буква) входния файл и да махаме всички тагове. Лесно се вижда, че всички тагове започват със символа "<" и завършват със символа ">". Това се отнася и за отварящите и за затварящите тагове. Това означава, че от всеки ред във файла трябва да се премахнат всички поднизове, започващи с "<" и завършващи с ">".

Проверка на идеята

Имаме идея за решаване на задачата. Дали идеята е вярна? Първо трябва да я проверим. Можем да я проверим дали е вярна за примерния входен файл, а след това да помислим дали няма някакви специални случаи, за които идеята би могла да е некоректна.

Взимаме лист и химикал и проверяваме на ръка идеята дали е вярна. Задраскваме всички поднизове от текста, които започват със символа "<" и завършват със символа ">". Като го направим, виждаме, че остава само чистият текст и всички тагове изчезват:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Сега остава да измислим някакви по-специални случаи. Нали не искаме да напишем 200 реда код и чак тогава да се сетим за тях и да трябва да преправяме цялата програма? Затова е важно да проверим проблемните ситуации, за които се сетим, още сега, преди да сме почнали да пишем кода на решението.

Можем да се сетим за следния специален пример:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

В него има две особености:

- Има тагове, съдържащи текст, които се отварят и затварят на различни редове.
- Има тагове, които съдържат хем текст, хем други тагове в себе си.

Какъв трябва да е резултатът за този пример? Ако директно махнем всички тагове, ще получим нещо такова:

```
Clickon this
linkfor more info.
This isboldtext.
```

Или може би трябва да следваме правилата на езика HTML и да получим следния текст:

```
Click on this link for more info.
This is bold text.
```

Има и други варианти, например да слагаме всяко парче текст, което не е таг, на нов ред:

```
Click
on this
link
for more info.
This is
bold
text.
```

Ако махнем всичкия текст в таговете и долепим останалия текст, ще получим думи, които са залепени една до друга. От условието на задачата не става ясно дали това е исканият резултат или трябва, както в езика HTML, да получим по един интервал между отделните тагове. В езика HTML всяка поредица от разделители (интервали, нов ред, табулации и др.) се визуализира като един интервал. Това, обаче, не е споменато в условието на задачата и не става ясно от примерния вход и изход.

Не става ясно още дали трябва да отпечатваме думите, които са в таг, съдържащ в себе си други тагове или да ги пропускаме. Ако отпечатваме само съдържанието на тагове, в които има единствено текст, ще получим нещо такова:

```
on this
link
bold
```

От условието не става ясно още как се визуализира текст, който е разположен на няколко реда във вътрешността на някой таг.

Изясняване на условието на задачата

Първото, което трябва да направим, когато открием неясен момент в условието на задачата, е да го прочетем внимателно. В случая условието наистина не е ясно и не ни дава отговор на въпросите. Най-вероятно не трябва да следваме HTML правилата, защото те не са описани в условието, но не става ясно дали долепяме думите в съседни тагове или си разделяме с нов ред.

Остава ни само едно: да питаме. Ако сме на изпит, ще питаме този, който ни е дал изпитните задачи. Ако сме в реалния живот, то все някой е поръчител на софтуера, който разработваме, и той би могъл да отговори на възникналите въпроси. Ако никой не може да отговори, избираме един от вариантите, който ни се струва най-правилен съгласно условието на задачата и действаме по него.

Приемаме, че трябва да се отпечата всичкия текст, който остава като премахнем всички отварящи и затварящи тагове, като използваме за разделител между отделните текстове празен ред. Ако в текста има празни редове, запазваме ги. За нашия пример трябва да получим следния изход:

```
Click
on this
link
for more info.
This is
bold
text.
```

Нова идея за решаване на задачата

И така, нашата адаптирана към новите изисквания идея е следната: четем файла ред по ред и във всеки ред заместваме таговете с нов ред. За да избегнем дублирането на нови редове в резултатния файл, заместваме всеки два последователни нови реда от резултата с един нов ред.

Проверяваме новата идея с оригиналния пример от условието на задачата и с нашия пример и се убеждаваме, че идеята този път е вярна. Остава да я реализираме.

Разбиваме задачата на подзадачи

Задачата лесно можем да разбием на подзадачи:

- Прочитане на входния файл.
- Обработка на един ред от входния файл: заместване на таговете със символ за нов ред.
- Записване на резултата в изходния файл.

Какво структури от данни да ползваме?

В тази задача трябва да извършваме проста текстообработка и работа с файлове. Въпросът какви структури от данни да ползваме не стои пред нас – за четене и писане във файл ще ползваме съответните класове от пакета **`java.io`**, а за текстообработката ще ползваме класа **`String`** и ако се наложи – **`StringBuilder`**.

Да помислим за ефективността

Ако четем редовете един по един, това няма да е бавна операция. Самата обработка на един ред може да се извърши чрез някакво заместване на символи с други – също бърза операция. Не би трябвало да имаме проблеми с производителността.

Може би проблеми ще създаде изчистването на празните редове. Ако събираме всички редове в някакъв буфер (**`StringBuilder`**) и след това премахваме двойните празни редове, този буфер ще заеме много памет при големи входни файлове (примерно при 500 MB входен файл).

За да спестим памет, ще се опитаме да чистим излишните празни редове още след заместване на таговете със символа за празен ред.

Вече разгледахме внимателно идеята за решаване на задачата, уверихме се, че е добра и хваща специалните случаи, които могат да възникнат, и смятаме, че няма да имаме проблеми с производителността. Сега вече можем спокойно да преминем към имплементация на алгоритъма. Ще пишем стъпка по стъпка, за да откриваме грешките възможно най-рано.

Стъпка 1 – прочитане на входния файл

Първата стъпка от решението на поставената задача е прочитането входния файл. В нашия случай той е HTML файл. Това не трябва да ни притеснява, тъй като HTML е текстов формат. Затова, за да го прочетем, ще използваме класа **`Scanner`**. Ще обходим входния файл ред по ред и за всеки ред ще извличаме (засега не ни интересува как) нужната ни информация (ако има) и ще я за писваме в един **`StringBuilder`**. Извличането ще реализираме в следващата стъпка (стъпка 2), а записването в някоя от по-следващите стъпки. Да напишем нужния код за реализацията на нашата първа стъпка:

```
Scanner scanner = new Scanner(new File("Problem1.html"));
while (scanner.hasNextLine()) {
   // Find what we need and save it in the result
}
scanner.close();
```

Чрез написания код ще прочетем входния файл ред по ред. Да помислим дали сме реализирали добре първата стъпка. Сещате ли се какво пропуснахме?

С написаното ще прочетем входния файл, но само ако съществува. Ами ако входния файл не съществува или не може да бъде отворен по някаква причина? Сегашното ни решение няма да се справи с този проблем. В кода има и още един проблем: ако настъпи грешка при четенето или обработката на данните от файла, той няма да бъде затворен.

За да избегнем тези проблеми трябва да използваме конструкцията **try-catch-finally**. Така, ако възникне изключение ще го обработим и накрая винаги ще затваряме файла, с които сме работили. Не трябва да забравяме, че обекта от **Scanner** трябва да е деклариран извън **try** блока, защото иначе ще е недостъпен във **finally** блока. Това не е фатална грешка, но често се допуска от начинаещите програмисти.

Добре е да дефинираме името на входния файл като константа, защото вероятно ще го ползваме на няколко места.

Още нещо: при четене от текстов файл е редно да зададем кодирането на файла. В случая ще използваме кодиране **windows-1251**.

Да видим до какво стигнахме:

```
import java.io.*;

public class HtmlTagRemover {
   private static final String INPUT_FILE_NAME =
      "Problem1.html";
   private static final String CHARSET = "windows-1251";

   public static void main(String args[]) {
      Scanner scanner = null;
      StringBuilder result = new StringBuilder();
      try {
         scanner = new Scanner(
            new File(INPUT_FILE_NAME), CHARSET);
         while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            // Process the next line here
         }
      } catch (IOException ioex) {
```

```
            System.err.println(
               "Can not read file " + INPUT_FILE_NAME + ".");
         } finally {
            if (scanner != null) {
               scanner.close();
            }
         }
      }
}
```

Справихме се с описаните проблеми и изглежда вече имаме коректно реализирано четенето на входния файл. За да сме напълно сигурни можем да тестваме. Например да изпишем съдържанието на входния файл на конзолата, а след това да пробваме с несъществуващ файл. Изписването ще става в **while** цикъла чрез **System.out.println(scanner.nextLine())**.

Ако тестваме с примера от условието на задачата, резултатът е следният:

```
<html>
<head>
<title>Welcome to our site!</title>
</head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a> -
<a href="/contenst.html">Contacts</a> -
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Сега, нека пробваме с несъществуващ файл. Да заменим името на файла **Problem1.html** с **Problem2.html**. Резултатът от това е следният:

```
File Problem2.html not found
```

Уверихме се, че дотук написаният код е верен. Да преминем към следващата стъпка.

Стъпка 2 – премахване на таговете

Сега трябва да измислим подходящ начин да премахнем всички тагове. Какъв да бъде начинът?

Един възможен начин е като проверяваме реда символ по символ. За всеки символ от текущия ред ще търсим символа "<". От него надясно ще знаем,

че е имаме някакъв таг (отварящ или затварящ). Краят на тага символът ">". Така можем да откриваме таговете и да ги премахваме. За да не получим долепяне на думите в съседни тагове, ще заместваме всеки таг със символа за празен ред "**\n**".

Алгоритъмът не е сложен за имплементиране, но дали няма по-хитър начин? Можем ли да използваме регулярни изрази? С тях лесно можем да търсим тагове и да ги заместваме с "**\n**", нали? Същевременно кодът няма да е сложен и при възникване на грешки по–лесно ще бъдат отстранени. Ще се спрем на този вариант. Какво трябва да направим? Първо трябва да напишем регулярния израз. Ето как изглежда той:

```
<[^>]*>
```

Идеята е проста: всеки низ, който започва с "**<**", продължава с произволи символи, различни от "**>**" и завършва с "**>**", е HTML таг. Ето как можем да заместим таговете със символ за нов ред:

```
private static String removeAllTags(String str) {
   String strWithoutTags = str.replaceAll("<[^>]*>", "\n");
   return strWithoutTags;
}
```

След като написахме тази стъпка, трябва да я тестваме. За целта отново ще изписваме намерените низове на конзолата чрез **System.out. println()**. Да тестваме кода, който получихме:

HtmlTagRemover.java

```
import java.io.*;
import java.util.*;

public class HtmlTagRemover {
   private static final String INPUT_FILE_NAME =
      "Problem1.html";
   private static final String CHARSET = "windows-1251";

   public static void main(String args[]) {
      Scanner scanner = null;
      try {
         scanner = new Scanner(
            new File(INPUT_FILE_NAME), CHARSET);
         while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            line = removeAllTags(line);
            System.out.println(line);
         }
      } catch (IOException ioex) {
```

```
            System.err.println("Can read file " + INPUT_FILE_NAME);
        } finally {
            if (scanner != null) {
                scanner.close();
            }
        }
    }

    private static String removeAllTags(String str) {
        String strWithoutTags = str.replaceAll("<[^>]*>", "\n");
        return strWithoutTags;
    }
}
```

Ако стартираме програмата за нашия специален пример, резултатът ще бъде е следният:

```
(празен ред)
Click
on this
link
for more info.
(празен ред)
This is
bold
text.
(празен ред)
```

Всичко е работи отлично, само, че имаме излишни празни редове. Можем ли да ги премахнем? Това ще е следващата ни стъпка.

Стъпка 3 – премахване на празните редове

Можем да премахнем излишните празни редове, като заменяме двоен празен ред "**\n\n**" с единичен празен ред "**\n**". Ето примерен метод, който извършва замяната:

```
private static String removeDoubleNewLines(String str) {
    String result = str.replaceAll("\n\n", "\n");
    return result;
}
```

Както, винаги, преди да продължим напред, тестваме метода дали работи коректно. Пробваме с текст, в който няма празни редове, а след това добавяме 2, 3, 4 и 5 празни реда, включително в началото и в края на текста.

Установяваме, че методът не работи коректно, когато има 4 празни реда един след друг. Например ако подадем като входни данни "**ab\n\n\n\ncd**", получаваме "**ab\n\n\cd**" вместо "**ab\ncd**". Този дефект се получава, защото **replaceAll()** намира и замества съвпаденията еднократно отляво надясно. Ако в резултат на заместване се появи отново търсеният низ, той бива прескочен.

Видяхте колко е полезно всеки метод да бъде тестван на момента, а не накрая да се чудим защо програмата не работи и да имаме 200 реда код, пълен с грешки. Ранното откриване на дефектите е много полезно и трябва да го правите винаги, когато е възможно. Ето поправения код:

```
private static String removeDoubleNewLines(String str) {
   while (str.indexOf("\n\n") != -1) {
      str = str.replaceAll("\n\n", "\n");
   }
   return str;
}
```

След серия тестове, се убеждаваме, че сега вече методът работи коректно. Сега можем да тестваме дали метод ни спасява от излишните нови редове. За целта правим следната промяна:

```
while (scanner.hasNextLine()) {
   String line = scanner.nextLine();
   line = removeAllTags(line);
   line = removeDoubleNewLines(line);
   System.out.println(line);
}
```

Изглежда пак има празни редове. От къде ли идват? Вероятно, ако имаме ред, който съдържа само тагове, той ще създаде проблем. Следователно трябва да предвидим този случай. Добавяме следната проверка:

```
if (! line.equals("\n")) {
   System.out.println(line);
}
```

Това ни спасява от повечето празни редове, но не и от всички.

Ако се замислим, би могло да се случи така, че някой ред да започва или завършва с таг. Тогава този таг ще бъде заменен с единичен празен ред и така в началото или в края на реда може да има празен ред. Това означава, че трябва да чистим празните редове в началото и в края на всеки ред. Ето как можем да направим въпросното изчистване:

```
private static String trimNewLines(String str) {
   int start = 0;
```

```
   while (start < str.length() && str.charAt(start)=='\n') {
      start++;
   }

   int end = str.length()-1;
   while (end >= 0 && str.charAt(end)=='\n') {
      end--;
   }

   if (start > end) {
      return "";
   }

   String trimmed = str.substring(start, end+1);
   return trimmed;
}
```

Методът работи много просто: преминава отляво надясно пред входния символен низ и прескача всички символи за празен ред. След това преминава отдясно наляво и отново прескача всички символи за празен ред. Ако лявата и дясната позиция са се разминали, това означава, че низът или е празен, или съдържа само символи за празен ред. Тогава връщаме празен низ. Иначе връщаме всичко надясно от стартовата позиция и наляво от крайната позиция.

Както винаги, тестваме въпросния метод дали работи коректно с няколко примера, сред които празен низ, низ без нови редове, низ с нови редове отляво или отдясно или и от двете страни и низ само с нови редове. Убеждаваме се, че методът работи коректно.

Сега остава да модифицираме логиката на обработката на входния файл:

```
while (scanner.hasNextLine()) {
   String line = scanner.nextLine();
   line = removeAllTags(line);
   line = removeDoubleNewLines(line);
   line = trimNewLines(line);
   if (! line.equals("")) {
      writer.println(line);
   }
}
```

Този път тестваме и се убеждаваме, че всичко работи коректно.

Стъпка 4 – записване на резултата във файл

Остава ни да запишем резултата в изходен файл. За да записваме резултата в изходния файл ще използваме **PrintStream**. Тази стъпка е тривиална.

Трябва да се съобразим само, че писането във файл може да предизвика изключение и затова трябва да променим леко логиката за обработка на грешки и за отварянето и затварянето на потоците за входния и изходния файл.

Ето какво се получава най-накрая като изход от програмата:

HtmlTagRemover.java

```
import java.io.*;
import java.util.*;

public class HtmlTagRemover {
   private static final String INPUT_FILE_NAME = "Problem1.html";
   private static final String OUTPUT_FILE_NAME = "Problem1.txt";
   private static final String CHARSET = "windows-1251";

   public static void main(String args[]) {

      Scanner scanner = null;
      PrintWriter writer = null;
      try {
         scanner = new Scanner(
            new File(INPUT_FILE_NAME), CHARSET);
         writer = new PrintWriter(OUTPUT_FILE_NAME, CHARSET);

         while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            line = removeAllTags(line);
            line = removeDoubleNewLines(line);
            line = trimNewLines(line);
            if (! line.equals("")) {
               writer.println(line);
            }
         }
      } catch (IOException ioex) {
         System.err.println("Can read or write file " + ioex);
      } finally {
         if (scanner != null) {
            scanner.close();
         }
         if (writer != null) {
            writer.close();
         }
      }
   }

   private static String removeAllTags(String str) {
```

```
      String strWithoutTags = str.replaceAll("<[^>]*>", "\n");
      return strWithoutTags;
   }

   private static String trimNewLines(String str) {
      int start = 0;
      while (start < str.length() && str.charAt(start)=='\n') {
         start++;
      }

      int end = str.length()-1;
      while (end >= 0 && str.charAt(end)=='\n') {
         end--;
      }

      if (start > end) {
         return "";
      }

      String trimmed = str.substring(start, end+1);
      return trimmed;
   }

   private static String removeDoubleNewLines(String str) {
      while (str.indexOf("\n\n") != -1) {
         str = str.replaceAll("\n\n", "\n");
      }
      return str;
   }
}
```

Тестване на решението

Досега тествахме отделните стъпки от решението на задачата. Чрез извършените тестове на отделните стъпки намаляваме възможността за грешки, но това не значи, че не трябва да тестваме цялото решение. Може да сме пропуснали нещо, нали?

Тестваме с примерния входен файл от условието на задачата. Всичко работи коректно.

Тестваме с нашия "сложен" пример. Всичко работи добре.

Задължително трябва да тестваме граничните случаи и да пуснем тест за производителност.

Започваме с празен файл. Изходът е коректен – празен файл.

Тестваме с файл, който съдържа само една дума "**Hello**" и не съдържа тагове. Резултатът е коректен – изходът съдържа само "**Hello**".

Тестваме с файл, който съдържа само тагове и не съдържа текст. Резултатът е отново коректен – празен файл.

Пробваме да сложим празни редове на най-невероятни места във входния файл. Пускаме следния тест:

```
Hello

<br><br>


<b>I<b> am here

I am not <b>here</b>
```

Изходът е следният:

```
Hello
I
 am here
I am not
here
```

Изглежда открихме дребен дефект. Има един интервал в началото на един от редовете. Според условието не е много ясно дали това е дефект, но нека се опитаме да го оправим.

Добавяме следния код при обработката на поредния ред от входния файл:

```
line = line.trim();
```

Дефектът не се премахва. Пускаме дебъгера и забелязваме защо се получава така. Причината е, че отпечатваме в изходния файл символен низ със стойност "**I\n am here**" и така получаваме интервал след празен ред. Можем да поправим дефекта, като навсякъде от празен ред, следван от празно пространство (празен ред, интервал, табулация и т.н.). Ето поправката:

```
private static String removeDoubleNewLines(String str) {
   str = str.replaceAll("\n\\s+", "\n");
   return str;
}
```

Направихме метода хем по-кратък, хем по-коректен. Единствено трябва да му сменим името с някакво по-адекватно, примерно нещо като **removeNewLinesWithWhiteSpace()**.

Сега трябва отново да тестваме упорито след поправката. Слагаме нови редове и интервали пръснати безразборно и се уверяваме се, че всичко работи вече коректно.

Остана един последен тест – за производителност. Лесно можем да създадем обемен входен файл. Дърпаме някой известен сайт, примерно http://java.sun.com/, взимаме му сорс кода и го копираме 1000 пъти. Получаваме достатъчно голям входен файл. В нашия случай се получи 44 MB файл с 947 000 реда. За обработката му бяха нужни под 10 секунди, което е напълно приемлива скорост.

Като надникнем в резултата, обаче, забелязваме много неприятен проблем. В него има части от тагове. По-точно виждаме следното:

```
<!--
var s_pageName="home page"
//-->
```

Бързо става ясно, че сме изпуснали един много интересен случай. В HTML може един таг да бъде затворен няколко реда след отварянето си, т.е. един таг може да е разположен на няколко последователни реда. Точно такъв е нашият случай: имаме таг с коментари, който съдържа JavaScript код. Ако програмата работеше коректно, щеше да отреже целия таг вместо да го запази в изходния файл.

Видяхте колко е полезно тестването и колко е важно. В някои сериозни фирми (като например Майкрософт) решение без тестове се счита за готово на 50%. Това означава, че ако пишете код 2 часа, трябва да отделите за тестване (ръчно или автоматизирано) поне още 2 часа! Само така можете да създадете качествен софтуер.

Колко жалко, че открихме проблема чак сега вместо в началото, когато проверявахме дали е правилна идеята ни за решение на задачата, прди да сме написали програмата. Понякога се случва така, нама как.

Как да оправим проблема с тагове на два реда?

Първата идея, която ни хрумва, е да заредим в паметта целия входен файл и да го обработваме като един голям стринг вместо ред по ред. Това е идея, която изглежда ще работи, но ще работи бавно и ще консумира голямо количество памет. Нека потърсим друга идея.

Очевидно не можем да четем файла ред по ред. Можем ли да го четем символ по символ? Ако можем, как ще обработваме таговете? Хрумва ни, че ако четем файла символ по символ, можем във всеки един момент да знаем дали сме в таг или сме извън таг и ако сме извън таг, можем да печатаме всичко, което прочетем. Ще се получи нещо такова:

```
boolean inTag = false;
```

```
while (! <end of file is reached>) {
   char ch = <read next character>
   if (ch == '<') {
      inTag = true;
   } else if (ch == '>') {
      inTag = false;
   } else {
      if (! inTag) {
         print(ch);
      }
   }
}
```

Идеята е много проста и лесна за реализация. Ако я реализираме директно, ще имаме проблема с празните редове и проблема със сливането на текст от съседни тагове. За да разрешим този проблем, можем да натрупваме текста в **StringBuilder** и да го отпечатваме при край на файла или при преминаване към таг. Ще се получи нещо такова:

```
boolean inTag = false;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>) {
   char ch = <read next character>
   if (ch == '<') {
      if (! inTag) {
         printBuffer(buffer);
      }
      buffer.clear();
      inTag = true;
   } else if (ch == '>') {
      inTag = false;
   } else {
      if (! inTag) {
         buffer.append(ch);
      }
   }
}
printBuffer(buffer);
```

Ако добавим и логиката за избягване на празните редове, както и четенето на входа и писането на резултата, ще получим цялостно решение на задачата по новия алгоритъм:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
```

```
public class SimpleHtmlTagRemover {
   private static final String INPUT_FILE_NAME = "Problem1.html";
   private static final String OUTPUT_FILE_NAME = "Problem1.txt";
   private static final String CHARSET = "windows-1251";

   public static void main(String[] args) throws IOException {
      InputStreamReader reader = new InputStreamReader(
           new FileInputStream(INPUT_FILE_NAME), CHARSET);
      PrintWriter writer = new PrintWriter(
        OUTPUT_FILE_NAME, CHARSET);
      try {
         boolean inTag = false;
         StringBuilder buffer = new StringBuilder();
         while (true) {
            int nextChar = reader.read();
            if (nextChar == -1) {
               // End of file reached
               printBuffer(writer, buffer);
               break;
            }
            char ch = (char) nextChar;
            if (ch == '<') {
               if (! inTag) {
                  printBuffer(writer, buffer);
               }
               buffer.setLength(0);
               inTag = true;
            } else if (ch == '>') {
               inTag = false;
            } else {
               // We have other character (not "<" or ">")
               if (! inTag) {
                  buffer.append(ch);
               }
            }
         }
      } finally {
         reader.close();
         writer.close();
      }
   }

   private static void printBuffer(PrintWriter writer,
        StringBuilder buffer) {
      String str = buffer.toString();
      String trimmed = str.trim();
      String textOnly = removeNewLineWithWhiteSpace(trimmed);
```

```
      if (textOnly.length() != 0) {
         writer.println(textOnly);
      }
   }

   private static String removeNewLineWithWhiteSpace(String str){
      str = str.replaceAll("\n\\s+", "\n");
      return str;
   }
}
```

За простота сме пропуснали обработката на грешки при четене и писане във файл. При възникване на изключение го изхвърляме от главния метод и оставяме виртуалната машина да го отпечата в конзолата.

Входният файл чете символ по символ с класа **InputStreamReader**. За съжаление не можем да ползваме любимият ни клас **Scanner**, защото той няма метод за четене на единичен символ.

Първоначално буферът за натрупване на текст е празен. В главния цикъл анализираме всеки прочетен символ. Имаме следните случаи:

- Ако стигнем до края на файла, отпечатваме каквото има в буфера и алгоритъмът приключва.
- При срещане на символ "**<**" (начало на таг) първо отпечатваме буфера (ако установим, че преминаваме от текст към таг). След това зачистваме буфера и установяваме **isTag = true**.
- При срещане на символ "**>**" (край на таг) установяваме **isTag = false**. Това ще позволи следващите след тага символи да се натрупват в буфера.
- При срещане на някой друг символ (текст или празно пространство), той се добава към буфера, ако сме извън таг. Ако сме в таг, символът се игнорира.

Печатането на буфера се грижи да премахва празните редове в текста и да изчиства празното пространство в началото и в края на текста. Как точно извършваме това, вече разгледахме в предходното решение на задачата, което се оказа грешно.

Тестване на новото решение

Остава да тестваме задълбочено новото решение. Изпълняваме всички тестове, които проведохме за предното решение. Добавяме тест с тагове, които се разпростират на няколко реда. Отново тестваме за производителност със сайта на Java 1000 пъти. Уверяваме се, че и за него програмата работи коректно и дори е по-бърза.

Най-накрая вече сме готови за следващата задача.

Задача 2: Лабиринт

Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо (**0**) или не (**x**). В едно от квадратчетата се намира нашият герой Минчо (*****):

x	x	x	x	x	x
0	x	0	0	0	x
x	*	0	x	0	x
x	x	x	x	0	x
0	0	0	0	0	x
0	x	x	x	0	x

Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Ако Минчо стъпи в клетка, която е на границата на лабиринта, той може с една стъпка да излезе извън него. Напишете програма, която по даден лабиринт отпечатва минималния брой стъпки, необходими на Минчо, за да излезе от лабиринта или -1 ако няма изход.

Входните данни се четат от текстов файл с име **Problem2.in**. На първия ред във файла стои числото N (2 < N < 100). На следващите N реда стоят по N символа, всеки от които е или "**0**" или "**x**" или "*****". Изходът представлява едно число и трябва да се изведе във файла **Problem2.out**.

Примерен входен файл **Problem2.in**:

```
6
xxxxxx
0x000x
x*0x0x
xxxx0x
00000x
0xxx0x
```

Примерен изходен файл **Problem2.out**:

```
9
```

Измисляне на идея за решение

Имаме лабиринт и трябва да намерим най-краткия път в него. Това не е лесна задача и трябва доста да помислим или да сме прочели някъде как се решават такива задачи.

Нашият алгоритъм ще започва от работата си от началната точка, която ни е дадена. Знаем, че можем да се предвижваме в съседна клетка хоризонтално и вертикално, но не и по диагонал. Нашият алгоритъм трябва да обхожда лабиринта по някакъв начин, за да намери в него най-късия път. Как да обхождаме клетките в лабиринта?

Един възможен начин за обхождане е следният: стартираме от началната клетка. Преместваме се в съседна клетка на текущата (която е проходима), след това в съседна клетка на нея (която е проходима и все още непосетена), след това в съседна на последната посетена (която е проходима е все още непосетена) и така продължаваме рекурсивно напред, докато или стигнем изход от лабиринта, или стигнем до място, от където няма продължение (няма съседна клетка, която е свободна и непосетена). В този момент се връщаме от рекурсията (към предходната клетка, от която сме стигнали текущата) и посещаваме друга клетка на предходната клетка. Ако няма продължение, се връщаме още назад. Описаният рекурсивен процес представлява обхождане на лабиринта в дълбочина (спомнете си главата "Рекурсия").

Възниква въпросът "Нужно ли е да минаваме през една клетка повече от един път"? Ако минаваме през една клетка най-много веднъж, то бързо ще обходим целия лабиринт и ако има изход, ще го намерим. Обаче минимален ли ще е този път. Ако си нарисуваме процеса на хартия, бързо ще се убедим, че намереният път няма да е минимален.

Ако при връщане от рекурсията отбелязваме като свободна клетката, която напускаме, ще позволим до една и съща клетка да стигаме многократно, идвайки по различен път. Пълното рекурсивно обхождане на лабиринта на практика ще намери всички възможни пътища от началната клетка до всяка други клетка. От всички тези пътища можем да вземем най-късия път до клетка на границата на лабиринта (изход) и така ще намерим решение на задачата.

Проверка на идеята

Изглежда имаме идея за решаване на задачата: с рекурсивно обхождане намираме всички пътища в лабиринта от началната клетка до клетка на границата на лабиринта и измежду всички тези пътища избираме най-късия. Нека да проверим идеята.

Взимаме лист хартия и си правим един примерен лабиринт. Пробваме алгоритъма. Вижда се, че той намира всички пътища от началната клетка до някой от изходите, като доста обикаля напред-назад. В крайна сметка намира всички изходи и измежду всички пътища може да се избере най-краткият.

Дали идеята работи, ако няма изход? Правим си втори лабиринт, който е без изход. Пробваме алгоритъма върху него, отново на лист хартия. Виждаме, че след доста обикаляне напред-назад алгоритъмът не намира нито един изход и приключва.

Изглежда имаме правилна идея за решаване на задачата. Да преминем напред и да помислим за структурите от данни.

Какви структури от данни да използваме?

Първо трябва да преценим как да съхраняваме лабиринта. Съвсем естествено е да ползваме матрица от символи, точно като тази на картинката. Ще считаме, че една клетка е проходима и можем да влезем в нея, ако съдържа символ, различен от символа '**x**'. Може да пазим лабиринта и в матрица с числа или булеви стойности, но разликата не е съществена. Матрицата от символи е удобна за отпечатване, а това ще ни помогне докато дебъгваме. Няма много възможности за избор. Ще съхраняваме лабиринта в матрица от символи.

След това трябва да решим в каква структура да запомняме обходените до момента клетки по време на рекурсията (текущия път). На нас ни трябва винаги последната обходена клетка. Това ни навежда на мисълта за структура, която спазва "последен влязъл, пръв излязъл", тоест стек. Можем да ползваме **`Stack<Cell>`**, където **`Cell`** е клас, съдържащ координатите на една клетка (номер на ред и номер на колона).

Остава да помислим в какво да запомняме намерените пътища, за да можем да извлечем накрая най-късия от тях. Ако се замислим малко, не е нужно да пазим всички пътища. Достатъчно е да помним текущия път и най-късият път за момента. Дори не е необходимо да пазим най-късия път за момента, ами само неговата дължина. Всеки път, когато намерим път до изход от лабиринта, можем да взимаме неговата дължина и ако тя е по-малка от най-късата дължина за момента, да я запомняме.

Изглежда намерихме ефективни структури от данни. Според препоръките за решаване на задачи, още не трябва да се втурваме да пишем кода на програмата, защото трябва да помислим за ефективността на алгоритъма.

Да помислим за ефективността

Нека да проверим идеята си от следна точка на ефективността? Какво правим ние? Намираме всички възможни пътища и от тях взимаме най-късия. Няма спор, че алгоритъмът ще работи, но ако лабиринтът стане много голям, дали ще работи бързо?

За да отговорим на този въпрос, трябва да помислим колко за пътищата. Ако вземем празен лабиринт, то на всяка стъпка на рекурсията ще имаме средно по 3 свободни продължения (като изключим клетката, от която идваме).

Така, ако имаме примерно лабиринт 10 на 10, пътят може да стане дълъг до 100 клетки и по време на обхождането на всяка стъпка ще имаме по 3 съседни клетки. Изглежда броят пътища е число от порядъка на 3 на степен 100. Очевидно алгоритъмът ще приспи компютъра много бързо.

Намерихме сериозен проблем на алгоритъма. Той ще работи много бавно, дори при малки лабиринти, а при големи изобщо няма да работи!

Да измислим нова идея

Разбрахме, че обхождането на всички пътища в лабиринта е неправилен подход, затова трябва да измислим друг.

Нека започнем от началната клетка и да обходим всички нейни съседни и да ги маркираме като обходени. За всяка обходена клетка ще запомняме едно число, което е равно на броя клетки, през които сме преминали, за да достигнем до нея (дължина на пътя от началната клетка до текущата).

За началната клетка дължината на пътя е 0. За нейните съседи дължината на пътя трябва да е 1, защото с 1 движение можем да ги достигнем от началната клетка. За съседните клетки на съседите на началната клетка дължината на пътя е 2. Можем да продължим да разсъждаваме по този начин и ще стигнем до следния алгоритъм:

1. Записваме дължина на пътя 0 за началната клетка. Отбелязваме я като посетена.
2. За всяка съседна клетка на началната отбелязваме, че пътят до нея е с дължина 1. Отбелязваме тези клетки като посетени.
3. За всяка клетка, която е съседна на клетка с дължина 1 и не е посетена, записваме, че е дължината на пътя до нея е 2. Отбелязваме въпросните клетки като посетени.
4. Продължавайки аналогично, на N-тата стъпка намираме всички непосетени все още клетки, които са на разстояние N премествания от началната клетка и ги отбелязваме като посетени.

Можем да визуализираме процеса по следния начин (взимаме друг лабиринт, за да покажем по-добра идеята):

Стъпка 0 – отбелязваме разстоянието от началната клетка до нея самата с 0 (за удобство на картинката отбелязваме свободните клетки с "-"):

x	x	x	x	x	x
-	x	-	-	-	x
x	0	-	x	-	x
x	-	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Стъпка 1 – отбелязваме с 1 всички проходими съседи на клетки със стойност 0:

x	x	x	x	x	x
-	x	-	-	-	x
x	0	1	x	-	x
x	1	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Стъпка 2 – отбелязваме с 2 всички проходими съседи на клетки със стойност 1:

x	x	x	x	x	x
-	x	2	-	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	-	-	-	x
-	x	x	x	-	x

Стъпка 3 – отбелязваме с 3 всички проходими съседи на клетки със стойност 2:

x	x	x	x	x	x
-	x	2	3	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	3	-	-	x
-	x	x	x	-	x

Продължавайки така, е ведно, че в един момент или ще достигнем клетка на границата на лабиринта (т.е. изход) или ще установим, че такава не е достижима.

Проверяване производителността на новия алгоритъм

Понеже никога не посещаваме повече от веднъж една и съща клетка, броят стъпки, които извършва този алгоритъм, не би трябвало да е голям. Примерно, ако имаме лабиринт с размери 100 на 100, той ще има 10 000 клетки, всяка от които ще посетим най-много веднъж и за всяка посетена

клетка ще проверим всеки неин съсед дали е свободен, т.е. ще извършим по 4 проверки. В крайна сметка ще извършим най-много 40 000 проверки и ще обходим най-много 10 000 клетки. Общо ще направим около 50 000 операции. Това означава, че алгоритъмът ще работи мигновено.

Проверяване коректността на новия алгоритъм

Изглежда този път нямаме проблем с производителността. Имаме бърз алгоритъм.

Да проверим дали е коректен. За целта си рисуваме на лист хартия някой по-голям и по-сложен пример, в който има много изходи и много пътища и започваме да изпълняваме алгоритъма. Изглежда работи коректно.

След това пробваме с лабиринт без изход. Изглежда алгоритъмът завършва, но не намира изход. Следователно работи коректно.

Пробваме още 2-3 примера и се убеждаваме, че този алгоритъм винаги намира най-краткия път до някой изход и винаги работи бързо, защото обхожда всяка клетка от лабиринта най-много веднъж.

Какви структури от данни да използваме?

С новия алгоритъм обхождаме последователно всички съседни клетки на началната клетка. Можем да ги сложим в някаква структура данни, примерно масив или по-добре списък, че в масива не можем да добавяме.

След това взимаме списъка с достигнатите на последната стъпка клетки и добавяме в друг списък техните съседи.

Така получаваме списък$_0$, който съдържа началната клетка, списък$_1$, който съдържа проходимите съседни на началната клетка, след това списък$_2$, който съдържа проходимите съседи на списък$_1$ и т.н. На **n**-тата стъпка получаваме списък$_n$, който съдържа всички клетки, достижими за точно **n** стъпки, т.е. клетките на разстояние **n** от стартовата клетка.

Изглежда можем да ползваме списък от списъци, за да пазим клетките, получени на всяка стъпка. Ако се замислим, за да получим **n**-тия списък, ни е достатъчен (**n-1**)-вия. Реално не ни трябва списък от списъци, а само списъкът от последната стъпка.

Можем да достигнем и до по-генерален извод: Клетките се обработват в реда на постъпване: когато свършват клетките от стъпка **k**, чак тогава се обработват клетките от стъпка **k+1**, а едва след тях – клетките от стъпка **k+2** и т.н. Процесът прилича на опашка – по-рано постъпилите клетки се обработват по-рано.

За да реализираме алгоритъма, можем да използваме опашка от клетки. За целта трябва да дефинираме клас клетка (**Cell**), който да съдържа координатите на дадена клетка (ред и колона). Можем да пазим в

матрицата за всяка клетка на какво разстояние се намира от началната клетка или -1, ако разстоянието още не е пресметнато.

Ако се замислим още малко, разстоянието от стартовата клетка може да се пази в самата клетка (в класа **Cell**) вместо да се прави специална матрица за разстоянията. Така ще се спести памет.

Вече имаме яснота за структурите данни. Остава да реализираме алгоритъма – стъпка по стъпка.

Стъпка 1 – класът Cell

Можем да започнем от дефиницията на класа **Cell**. Той ще ни трябва, за да запазим стартовата клетка, от която започва търсенето на пътя:

```
private class Cell {
   int row;
   int col;
   int distance;
}
```

Може да му добавим и конструктор за удобство:

```
public Cell(int row, int col, int distance) {
   this.row = row;
   this.col = col;
   this.distance = distance;
}
```

Стъпка 2 – прочитане на входния файл

Ще четем входния файл ред по ред чрез познатия ни клас **Scanner**. На всеки ред ще анализираме символите и ще ги записваме в матрица от символи. При достигане на символ "*****" ще запомним координатите му в инстанция на класа **Cell**, за да знаем от къде да започнем търсенето на най-краткия път за излизане от лабиринта.

Можем да дефинираме клас **Maze** и в него да пазим матрицата на лабиринта и стартовата клетка:

Maze.java

```
public class Maze {
   private char[][] maze;
   private int size;
   private Cell startCell = null;

   public void readFromFile(String fileName)
```

```
        throws FileNotFoundException {
      Scanner scanner = new Scanner(new File(fileName));
      try {
         // Read maze size
         this.size = scanner.nextInt();
         scanner.nextLine();

         // Create the maze
         this.maze = new char[this.size][this.size];

         // Read the maze cells from the file
         for (int row=0; row<this.size; row++) {
            String line = scanner.nextLine();
            for (int col = 0; col < line.length(); col++) {
               char ch = line.charAt(col);
               maze[row][col] = ch;
               if (ch == '*') {
                  this.startCell = new Cell(row, col, 0);
               }
            }
         }
      } finally {
         scanner.close();
      }
   }
}
```

Вече имаме класа **Maze** и подходящо представяне на данните от входния файл. За да сме сигурни, че написаното дотук е вярно трябва да тестваме. Можем да проверим дали матрицата е вярно попълнена, като я отпечатаме на конзолата. Друг вариант е да се разгледат стойностите на полетата от класа **Maze** през дебъгера на Eclipse.

След като тестваме написаното дотук продължаваме със следващата стъпка, а именно търсенето на най-краткия път.

Стъпка 3 – намиране на най-къс път

Можем директно да имплементираме алгоритъма, който вече дискутирахме. Трябва да дефинираме опашка и в нея да сложим в началото стартовата клетка. След това в цикъл трябва да взимаме поредната клетка от опашката и да добавяме всичките ѝ непосетени проходими съседи. На всяка стъпка има шанс да стъпим в клетка от границата на лабиринта, при което считаме, че сме намерили изход и търсенето приключва. Повтаряме цикъла докато опашката свърши. При всяко влизане посещение на дадена клетка проверяваме дали клетката е свободна и ако е свободна, я маркираме като непроходима. Така избягваме повторно попадане в същата клетка. Ето как изглежда имплементацията на алгоритъма:

```
public int findShortestPath(Cell startCell) {
   // Queue for traversing the cells in the maze
   Queue<Cell> visitedCells = new LinkedList<Cell>();
   visitCell(visitedCells, startCell.row, startCell.col, 0);

   // Perform Breath-First-Search (BFS)
   while (! visitedCells.isEmpty()) {
      Cell currentCell = visitedCells.remove();
      int row = currentCell.row;
      int col = currentCell.col;
      int distance = currentCell.distance;
      if ((row == 0) || (row == size-1)
         || (col == 0) || (col == size-1)) {
         // We are at the maze border
         return distance + 1;
      }
      visitCell(visitedCells, row, col + 1, distance + 1);
      visitCell(visitedCells, row, col - 1, distance + 1);
      visitCell(visitedCells, row + 1, col, distance + 1);
      visitCell(visitedCells, row - 1, col, distance + 1);
   }

   // We didn't reach any cell at the maze border -> no path
   return -1;
}

private void visitCell(Queue<Cell> visitedCells,
      int row, int col, int distance) {
   if (maze[row][col] != 'x') {
      // Cell is free. Visit it
      maze[row][col] = 'x';
      Cell cell = new Cell(row, col, distance);
      visitedCells.add(cell);
   }
}
```

Проверка след стъпка 3

Преди да се захванем със следващата стъпка, трябва да тестваме, за да проверим нашия алгоритъм. Трябва да пробваме нормалния случай, както и граничните случаи, когато няма изход, когато се намираме на изход, когато входният файл не съществува или квадратната матрица е с размер нула. Едва след това може да започнем решаването на следващата стъпка.

Нека да пробваме случая, в който имаме дължина нула на квадратната матрица във входния файл:

```
Exception in thread "main" java.lang.NullPointerException
```

```
   at Maze.findShortestPath(Maze.java:59)
   at Maze.main(Maze.java:104)
```

Допуснали сме грешка. Проблемът е в това, че при създаване на обект от класа **Maze**, променливата, в която ще помним началната клетка, се инициализира с **null**. Ако лабиринтът няма клетки (дължина 0) или липсва стартовата клетка, би трябвало програмата да връща резултат -1, а не да дава изключение. Можем да добавим проверка в началото на метода **findShortestPath()**:

```
public int findShortestPath(Cell startCell) {
   if (startCell == null) {
      // Start cell is missing -> no path
      return -1;
   }
   …
```

В останалите случаи изглежда, че алгоритъмът работи.

Стъпка 4 – записване на резултата във файл

Остава да запишем резултата от метода **FindShortestWay()** в изходния файл. Това е тривиална задача:

```
public void saveResult(String fileName, int result)
      throws IOException {
   FileWriter writer = new FileWriter(fileName);
   try {
      writer.write("" + result);
   } finally {
      writer.close();
   }
}
```

Ето как изглежда пълният код на решението на задачата:

Maze.java

```
import java.io.*;
import java.util.*;

public class Maze {
   private static final String INPUT_FILE_NAME = "Problem2.in";
   private static final String OUTPUT_FILE_NAME = "Problem2.out";

   private class Cell {
      int row;
```

```
    int col;
    int distance;

    public Cell(int row, int col, int distance) {
      this.row = row;
      this.col = col;
      this.distance = distance;
    }
  }

  private char[][] maze;
  private int size;
  private Cell startCell = null;

  public void readFromFile(String fileName)
      throws FileNotFoundException {
    Scanner scanner = new Scanner(new File(fileName));
    try {
      // Read maze size
      this.size = scanner.nextInt();
      scanner.nextLine();

      // Create the maze
      this.maze = new char[this.size][this.size];

      // Read the maze cells from the file
      for (int row=0; row<this.size; row++) {
        String line = scanner.nextLine();
        for (int col = 0; col < line.length(); col++) {
          char ch = line.charAt(col);
          maze[row][col] = ch;
          if (ch == '*') {
            this.startCell = new Cell(row, col, 0);
          }
        }
      }
    } finally {
      scanner.close();
    }
  }

  public int findShortestPath(Cell startCell) {
    if (startCell == null) {
      // Start cell is missing -> no path
      return -1;
    }

    // Queue for traversing the cells in the maze
```

```
    Queue<Cell> visitedCells = new LinkedList<Cell>();
    visitCell(visitedCells, startCell.row, startCell.col, 0);

    // Perform Breath-First-Search (BFS)
    while (! visitedCells.isEmpty()) {
      Cell currentCell = visitedCells.remove();
      int row = currentCell.row;
      int col = currentCell.col;
      int distance = currentCell.distance;
      if ((row == 0) || (row == size-1)
        || (col == 0) || (col == size-1)) {
        // We are at the maze border
        return distance + 1;
      }
      visitCell(visitedCells, row, col + 1, distance + 1);
      visitCell(visitedCells, row, col - 1, distance + 1);
      visitCell(visitedCells, row + 1, col, distance + 1);
      visitCell(visitedCells, row - 1, col, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
  }

  private void visitCell(Queue<Cell> visitedCells,
      int row, int col, int distance) {
    if (maze[row][col] != 'x') {
      // Cell is free. Visit it
      maze[row][col] = 'x';
      Cell cell = new Cell(row, col, distance);
      visitedCells.add(cell);
    }
  }

  public void saveResult(String fileName, int result)
      throws IOException {
    FileWriter writer = new FileWriter(fileName);
    try {
      writer.write("" + result);
    } finally {
      writer.close();
    }
  }

  public static void main(String[] args) throws IOException {
    Maze maze = new Maze();
    maze.readFromFile(INPUT_FILE_NAME);
    int pathLength = maze.findShortestPath(maze.startCell);
```

```
      maze.saveResult(OUTPUT_FILE_NAME, pathLength);
   }
}
```

Тестване на решението на задачата

След като имаме решение на задачата трябва да тестваме. Вече тествахме граничните случаи и случаи като липса на изход или началната позиция да е на изхода. Видяхме, че алгоритъмът работи коректно.

Остава да тестваме с голям лабиринт, например 1000 на 1000. Можем да си направим такъв лабиринт много лесно – с copy/paste. Изпълняваме теста и се убеждаваме, че програмата работи коректно за големия тест и работи изключително бързо – не се усеща каквото и да е забавяне.

При тестването трябва да се опитваме по всякакъв начин да счупим нашето решение. Пускаме още няколко по-трудни примера (примерно лабиринт с проходими клетки във формата на спирала). Можем да сложим голям лабиринт с много пътища, но без изход. Можем да сложим и каквото още се сетим.

Накрая се убеждаваме, че имаме коректно решение и преминаваме към следващата задача.

Задача 3: Магазин за авточасти

Фирма планира създаване на система за управление на магазин за авточасти. Една част може да се използва при различни модели автомобили и има следните характеристики:

Код, наименование, категория (за ходовата част, гуми и джанти, за двигателя, аксесоари и т.н.), покупна цена, продажна цена, списък с модели автомобили, за които може да се използва (даден автомобил се описва с марка, модел и година на производство, примерно BMW 316i, 1992), фирма-производител.

Фирмите-производители се описват с наименование, държава, адрес, телефон и факс.

Да се проектира съвкупност от класове с връзки между тях, които моделират данните за магазина. Да се напише демонстрационна програма, която показва коректната работа на всички класове.

Измисляне на идея за решение

От нас се изисква да създадем съвкупност от класове и връзки между тях, които да описват данните за магазина. Трябва да разберем кои съществителни са важни за решаването на задачата. Те са обекти от реалния свят, на които съответстват класове.

Кои са тези съществителни, които ни интересуват? Имаме магазин, авточасти, автомобили и фирми-производители. Трябва да създадем клас описващ магазин. Той ще се казва **Shop**. Другите класове съответно са **Part**, **Car** и **Manufacturer**. В условието на задачата има и други съществителни, например код на една част или година на производство на дадена кола. За тези съществителни няма да създаваме отделни класове, защото можем да използваме примитивните типове в Java. Това означава, че в класа **Part** ще има примерно поле **code** от тип **String**.

Вече знаем кои ще са нашите класове, както и полетата, които ги описват. Остава да си изясним връзките между обектите.

Каква структури от данни да използване, за да опишем връзката между два класа?

За да опишем връзката между два класа можем да използваме масив. При масива имаме достъп до елементите му по индекс, но веднъж след като го създадем не можем да му променяме дължината. Това го прави неудобен за нашата задача, понеже не знаем колко части ще имаме в магазина и по всяко време може да докарат още части или някой да купи някоя част и да се наложи да я изтрием или променим.

По-удобен е **ArrayList<T>**. Той притежава предимствата на масив, а освен това е с променлива дължина и с него лесно се реализира въвеждане и изтриване на елементи.

Засега изглежда, че **ArrayList<T>** е най-подходящ. За да се убедим ще разгледаме още няколко структури от данни. Например хеш-таблица – не е удобна в този случаи, понеже структурата "части" не от типа ключ-стойност. Тя би била подходяща, ако в магазина всяка част има уникален номер (например баркод). Тогава ще можем да ги търсим по този уникален номер. Структури като стек и опашка са неуместни.

Структурата "множество" и нейните имплементации **TreeSet** и **HashSet** се ползват, когато имаме уникалност по даден ключ. Може би на места, ще е добра да ползваме тази структура, за да избегнем повторения. Трябва да имаме предвид, че ползването на **HashSet<T>** изисква да имаме методи **hashCode()** и **equals()**, дефинирани коректно в типа **T**.

В крайна сметка избираме да ползваме **ArrayList<T>** и **HashSet<T>**.

Разделяне на задачата на подзадачи

Сега остава да си изясним въпроса от къде да започнем написването на задачата. Ако започнем да пишем класа **Shop**, ще се нуждаем от класа **Part**. Това ни подсеща, че трябва да започнем от клас, който не зависи от другите. Ще разделим написването на всеки клас на подзадача, като ще започнем от независещите от другите класове:

- Клас описващ автомобил – **Car**

- Клас описващ производител на части – **Manufacturer**
- Клас описващ част за автомобили – **Part**
- Клас за магазина – **Shop**
- Клас за тестване на останалите класове с примерни данни – **TestShop**

Имплементиране: стъпка по стъпка

Започваме написването на класовете, които сме описали в нашата идея. Ще ги създаваме в реда, по който са изброени в списъка.

Стъпка 1: класът Car

Започваме решаването на задачата с дефинирането на класа **Car**. В дефиницията имаме три полета, които показват производителя, модела и годината на производство на една кола и стандартния метод **toString()**, който връща низ с информация за дадена кола. Дефинираме го по следния начин:

Car.java

```
public class Car {
   private String brand;
   private String model;
   private String productionYear;

   public Car(String brand, String model, String productionYear){
      this.brand = brand;
      this.model = model;
      this.productionYear = productionYear;
   }

   @Override
   public String toString() {
      return "<" + this.brand + "," + this.model + ","
      + this.productionYear + ">";
   }
}
```

Стъпка 2: класът Manufacturer

Следва да реализираме дефиницията на класа **Manufacturer**, който описва производителя на дадена част. Той ще има пет полета – име, държава, адрес, телефонен номер и факс. Класът ще има и два метода – **getName()** и стандартния метод **toString()**. Първият ще връща низ с името за даден производител, а вторият – цялата информация за него.

Manufacturer.java

```
public class Manufacturer {
   private String name;
   private String country;
   private String address;
   private String phoneNumber;
   private String fax;

   public Manufacturer(String name,String country,
        String address, String phoneNumber, String fax) {
      this.name = name;
      this.country = country;
      this.address = address;
      this.phoneNumber = phoneNumber;
      this.fax = fax;
   }

   public String getName(){
      return this.name;
   }

   @Override
   public String toString(){
      return this.name + " <"+this.country+ "," + this.address +
         "," + this.phoneNumber + "," + this.fax + ">";
   }
}
```

Стъпка 3: класът Part

Сега трябва да дефинираме класа **Part**. Дефиницията му ще включва следните полета – име, код, категория, списък с коли, с които може да се използва дадената част, начална и крайна цена и производител. Тук вече ще използваме избраната от нас структура от данни **ArrayList<T>**. В случая ще бъде **ArrayList<Car>**. Полето показващо производителя на частта ще бъде от тип **Manufacturer**, защото задача изисква да се помни допълнителна информация за производителя. Ако се искаше да се знае само името на производителя (както случая с класа Car) нямаше да има нужда от този клас. Щяхме да имаме поле от тип **String**. За полето, което описва категорията на частта ще използваме **enum**:

PartCategory.java

```
public enum PartCategory {
   ENGINE, TIRES, EXHAUST, SUSPENSION, BRAKES
}
```

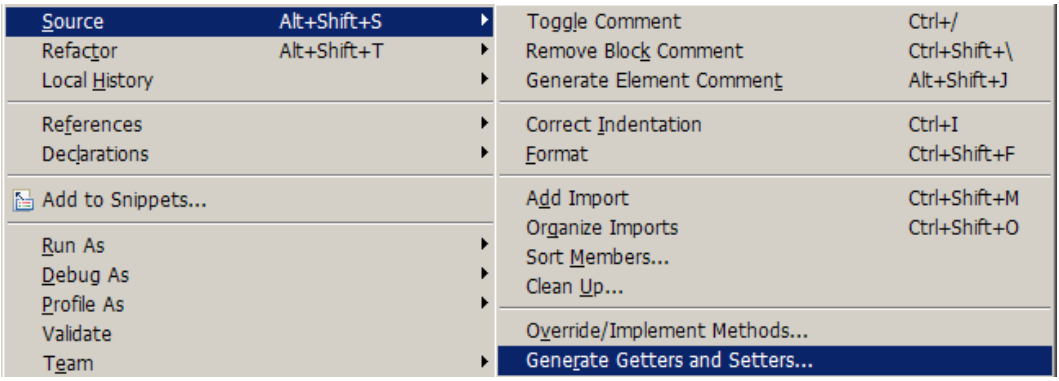
Ще създадем и метод, който ще отпечатва на конзолата полетата. Той ще се казва **printParts(PrintStream output)**. Нужен ни е метод за добавяне на кола (обект от тип **Car**) в списъка с колите (в **HashSet<Car>**). Той ще се казва **addSupportedCar(Car car)**. Ето го и кода на класа **Part**:

Part.java

```
import java.util.HashSet;

public class Part {
   private String name;
   private String code;
   private PartCategory category;
   private HashSet<Car> supportedCars;
   private double buyPrice;
   private double sellPrice;
   private Manufacturer manufacturer;

   public Part(String name, double buyPrice, double sellPrice,
           Manufacturer manufacturer, String code,
           PartCategory category) {
      this.name = name;
      this.buyPrice = buyPrice;
      this.sellPrice = sellPrice;
      this.manufacturer = manufacturer;
      this.code = code;
      this.category = category;
      this.supportedCars = new HashSet<Car>();
   }

   public void addSupportedCar(Car car) {
      this.supportedCars.add(car);
   }

   @Override
   public String toString() {
      StringBuilder result = new StringBuilder();
      result.append("Part: " + this.name + "\n");
      result.append("-code: " + this.code + "\n");
      result.append("-category: " + this.category + "\n");
      result.append("-buyPrice: " + this.buyPrice + "\n");
      result.append("-sellPrice: " + this.sellPrice + "\n");
      result.append("-manufacturer: " + this.manufacturer +"\n");
      result.append("---Supported cars---" + "\n");
      for (Car car:this.supportedCars) {
         result.append(car);
         result.append("\n");
      }
```

```
        result.append("----------------------\n");
        return result.toString();
    }
}
```

Понеже ползваме **HashSet<Car>** е необходимо да дефинираме методите **hashCode()** и **equals()**. За по-лесно ползваме функцията в Eclipse за автоматично генериране на код, достъпна от контекстното меню (Source → Generate hashCode() и equals()…):



Получаваме следния автоматично генериран код:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((brand == null) ? 0 :
        brand.hashCode());
    result = prime * result + ((model == null) ? 0 :
        model.hashCode());
    result = prime * result + ((productionYear == null) ? 0 :
        productionYear.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Car other = (Car) obj;
    if (brand == null) {
```

```
      if (other.brand != null)
         return false;
   } else if (!brand.equals(other.brand))
      return false;
   if (model == null) {
      if (other.model != null)
         return false;
   } else if (!model.equals(other.model))
      return false;
   if (productionYear == null) {
      if (other.productionYear != null)
         return false;
   } else if (!productionYear.equals(other.productionYear))
      return false;
   return true;
}
```

Стъпка 4: класът Shop

Вече имаме всички нужни класове за създаване на класа **Shop**. Той ще има две полета – име и списък от части, които се продават. Списъкът ще бъде **ArrayList<Part>**. Ще си добавим и два метода **addPart(Part part)** и **toString()**. Чрез първия ще добавяме нова част, а чрез втория ще отпечатаме името на магазина и частите в него. Ето примерна реализация:

Shop.java

```
import java.util.ArrayList;

public class Shop {
   private String name;
   private ArrayList<Part> parts;

   public Shop(String name){
      this.name = name;
      parts = new ArrayList<Part>();
   }

   public void addPart(Part part){
      parts.add(part);
   }

   @Override
   public String toString() {
      StringBuilder result = new StringBuilder();
      result.append("Shop: " + this.name + "\n\n");
      for(Part part : parts) {
```

```
            result.append(part);
            result.append("\n");
        }
        return result.toString();
    }
}
```

Стъпка 5: класът TestShop

Създадохме всички нужни класове. Остава да създадем още един, с който да демонстрираме използването на всички останали класове. Той ще се казва **TestShop**. В **main()** метода ще създадем два производителя и няколко коли. Ще ги добавим към две части. Частите ще добавим към обект от тип **Shop**. Накрая ще отпечатаме всичко на конзолата. Ето примерния код:

TestShop.java

```
public class TestShop {
   public static void main(String args[]) {
      Manufacturer bmw = new Manufacturer("BWM",
         "Germany", "Bavaria", "665544", "876666");
      Manufacturer lada = new Manufacturer("Lada",
         "Russia", "Moscow", "653443", "893321");

      Car bmw316i = new Car("BMW", "316i", "1994");
      Car ladaSamara = new Car("Lada", "Samara", "1987");
      Car mazdaMX5 = new Car("Mazda", "MX5", "1999");
      Car mercedesC500 = new Car("Mercedes", "C500", "2008");
      Car trabant = new Car("Trabant", "super", "1966");
      Car opelAstra = new Car("Opel", "Astra", "1997");

      Part cheapPart = new Part("Tires 165/50/13", 302.36,
         345.58, lada, "T332", PartCategory.TIRES);
      cheapPart.addSupportedCar(ladaSamara);
      cheapPart.addSupportedCar(trabant);

      Part expensivePart = new Part("BMW Engine Oil",
         633.17, 670.0, bmw, "Oil431", PartCategory.ENGINE);
      expensivePart.addSupportedCar(bmw316i);
      expensivePart.addSupportedCar(mazdaMX5);
      expensivePart.addSupportedCar(mercedesC500);
      expensivePart.addSupportedCar(opelAstra);

      Shop newShop = new Shop("Tunning shop");
      newShop.addPart(cheapPart);
      newShop.addPart(expensivePart);
```

```
      System.out.println(newShop);
   }
}
```

Това е резултатът от изпълнението на нашата програма:

```
Shop: Tunning shop

Part: Tires 165/50/13
-code: T332
-category: TIRES
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
<Lada,Samara,1987>
<Trabant,super,1966>
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: ENGINE
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<Opel,Astra,1997>
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
----------------------
```

Тестване на решението

Накрая остава да тестваме нашата задача. Всъщност ние направихме това с класа **TestShop**. Това обаче не означава, че сме изтествали напълно нашата задача. Трябва да се проверят граничните случаи, например когато някои от списъците са празни. Да променим малко кода в **main()** метода, за да пуснем задачата с празен списък:

TestShop.java

```
public class TestShop {
   public static void main(String args[]) {
      Manufacturer bmw = new Manufacturer("BWM",
         "Germany", "Bavaria", "665544", "876666");
      Manufacturer lada = new Manufacturer("Lada",
```

```
        "Russia", "Moscow", "653443", "893321");

    Car bmw316i = new Car("BMW", "316i", "1994");
    Car ladaSamara = new Car("Lada", "Samara", "1987");
    Car mazdaMX5 = new Car("Mazda", "MX5", "1999");
    Car mercedesC500 = new Car("Mercedes", "C500", "2008");
    Car trabant = new Car("Trabant", "super", "1966");
    Car opelAstra = new Car("Opel", "Astra", "1997");

    Part cheapPart = new Part("Tires 165/50/13", 302.36,
       345.58, lada, "T332", PartCategory.TIRES);

    Part expensivePart = new Part("BMW Engine Oil",
       633.17, 670.0, bmw, "Oil431", PartCategory.ENGINE);
    expensivePart.addSupportedCar(bmw316i);
    expensivePart.addSupportedCar(mazdaMX5);
    expensivePart.addSupportedCar(mercedesC500);
    expensivePart.addSupportedCar(opelAstra);

    Shop newShop = new Shop("Tunning shop");
    newShop.addPart(cheapPart);
    newShop.addPart(expensivePart);

    System.out.println(newShop);
  }
}
```

Резултатът от този тест е следният:

```
Shop: Tunning shop

Part: Tires 165/50/13
-code: T332
-category: TIRES
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: ENGINE
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
```

```
<Opel,Astra,1997>
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
----------------------
```

От резултата се вижда, че списъкът от коли на евтината част е празен. Това е и правилният изход. Следователно нашата задача изпълнява коректно граничния случай с празен списък.

Упражнения

1. Даден входен файл **mails.txt**, който съдържа имена на потребители и техните email адреси. Всеки ред от файла изглежда така:

   ```
   <first name> <last name> <username>@<host>.<domain>
   ```

   Има изискване за имейл адресите – **<username>** може да е последователност от латински букви (**a**-**z**, **A**-**Z**) и долна черна (_), **< host>** е последователност от малки латински букви (**a**-**z**), а **<domain>** има ограничение от 2 до 4 малки латински букви (**a**-**z**). Да се напише програма, която намира валидните email адреси и ги записва заедно с имената на потребителите в изходен файл **validMails.txt**.

2. Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо (0) или не (x):

x	x	x	0	x	x
0	x	0	0	0	
0	*	0	x	0	0
x	x	x	x	0	x
0	0	0	0	0	x
0	x	0	x	x	0

   В едно от квадратчетата се намира отново нашият герой Минчо (*). Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Напишете програма, която по даден лабиринт отпечатва броя на възможните изходи от лабиринта.

   Входните данни се четат от текстов файл с име **Problem.in**. На първия ред във файла стои числото N (2 < N < 1000). На следващите N реда стоят по N символа, всеки от които е или "**0**" или "**x**" или "*****". Изходът представлява едно число и трябва да се изведе във файла **Problem.out**.

3. Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо или не. Проходимите клетки съдържат малка латинска буква между "**a**" и "**z**", а непроходимите – '**#**'. В едно от квадратчетата се намира Минчо. То е означено с "*****".

   Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Когато Минчо минава през проходимите квадратчета, той си записва буквите от всяко квадратче. На всеки изход получава дума. Напишете програма, която по даден лабиринт отпечатва думите, които се образуват при всички възможни изходи от лабиринта.

a	#	#	k	m	#
z	#	a	d	a	#
a	*	m	#	#	#
#	d	#	#	#	#
r	i	f	i	d	#
#	d	#	d	#	t

   Входните данни се четат от текстов файл с име **Problem.in**. На първия ред във файла стои числото N (2 < N < 10). На следващите N реда стоят по N символа, всеки от които е или "**0**" или "**#**" или "*****". Изходът представлява едно число и трябва да се изведе във файла **Problem.out**.

4. Фирма планира създаване на система за управление на звукозаписна компания. Звукозаписната компания има име, адрес, собственик и изпълнители. Всеки изпълнител има име, псевдоним и създадени албуми. Албумите се описват с име, жанр, година на издаване, брой на продадените копия и списък от песни. Песните, от своя страна се описват с име и времетраене. Да се проектира съвкупност от класове с връзки между тях, които моделират данните за звукозаписната компания. Да се реализира тестов клас, който демонстрира работата на всички останали класове.

5. Фирма планира създаване на система за управление на компания за недвижими имоти. Компанията има име, собственик, Булстат, служители и разполага със списък от имоти за продажба. Служители се описват с име, длъжност и стаж. Компанията продава няколко вида имоти – апартаменти, къщи, незастроени площи и магазини. Всички те се характеризират с площ, цена на квадратен метър и местоположение. За някои от тях има допълнителна информация. За апартамента има данни за номер на етажа, дали в блока има асансьор и дали е обзаведен. За къщите се зная квадратните метри на застроена част и на незастроената (двора), на колко етажа е и дали е обзаведена. Да се проектира съвкупност от класове с връзки между тях, които моделират данните за компанията. Да се реализира тестов клас, който демонстрира работата на всички останали класове.

Решения и упътвания

1. Задачата е подобна на първата от примерния изпит. Отново трябва да чете ред по ред от входния файл и чрез подходящ регулярен израз да извличате имейл адресите.

   Примерен входен файл:

   ```
   Ivan Dimitrov ivan_dimitrov@abv.bg
   Svetlana Todorova Svetlana_tv@mail.bg
   Kiril Kalchev kalchev@gmail.com
   Todor Ivanov todo*r@888.com
   Ivelina Petrova ivel&7@abv.bg
   Petar Petrov pesho<5.mail.bg
   ```

   Изходен файл:

   ```
   Ivan Dimitrov ivan_dimitrov@abv.bg
   Svetlana Todorova Svetlana_tv@mail.bg
   Kiril Kalchev kalchev@gmail.com
   ```

   Тествайте внимателно решението си преди да преминете към следващата задача.

2. Възможните изходи от лабиринта са всички клетки, които се намират на границата на лабиринта и са достижими от стартовата клетка. Задачата се решава с дребна модификация на решението на задачата за лабиринта.

3. Задачата е изглежда подобна на предната, но се искат всички възможни пътища до изхода. Можете да направите рекурсивно търсене с връщане назад (backtracking) и да натрупвате в **StringBuilder** буквите до изхода, за да образувате думите, които трябва да се отпечатат. При големи лабиринти задачата няма решение (защото се използва пълно изчерпване и броят пътища до някой от изходите може да е ужасно голям).

4. Трябва да напишете нужните класове – **MusicCompany**, **Singer**, **Album**, **Song**. Помислете за връзките между класовете и какви структури данни да ползвате за тях. За отпечатването припокривайте метода **toString()** от **java.lang.Object**. Тествайте всички методи и граничните случаи.

5. Класовете, които трябва да напишете са **EstateCompany**, **Employee**, **Apartment**, **House**, **Shop** и **Area**. Забележете, че класовете, които ще описват недвижимите имоти имат някои еднакви характеристики. Изнесете тези характеристики в базов отделен клас **Estate**. Създайте метод **toString()**, който да изписва на конзолата данните от този клас. Пренапишете метода за класовете, които наследяват този клас, за да

показва цялата информация за всеки клас. Тествайте всички методи и граничните случаи.

Глава 24. Примерен изпит по програмиране – 8.04.2006 г.

Автор

Радослав Иванов

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на няколко примерни задачи от изпит по програмиране в НАРС, проведен на 8.04.2006 г. При решаването на задачите ще се придържаме към съветите от темата "Как да решаваме задачи по програмиране" и ще онагледим прилагането им в практиката.

Задача 1: Броене на думи в текст

Напишете програма, която преброява думите в даден текст, който се въвежда от конзолата. Програмата трябва да извежда общия брой думи, броя думи, изписани изцяло с главни букви и броя думи, изписани изцяло с малки букви. Ако дадена дума се среща няколко пъти на различни места в текста, всяко срещане се брои като отделна дума. За разделител между думите се счита всеки символ, който не е буква.

Примерен вход:

```
Добре дошли в Софтуерния университет (СОФТУНИ)!
```

Примерен изход:

```
Общо думи: 6
Думи с главни букви: 1
Думи с малки букви: 3
```

Намиране на подходяща идея за решение

Интуитивно ни идва наум, че можем да решим задачата, като разделим текста на отделни думи и след това преброим тези, които ни интересуват.

Тази идея очевидно е вярна, но е прекалено обща и не ни дава конкретен метод за решаването на проблема. Нека се опитаме да я конкретизираме и да проверим дали е възможно чрез нея да реализираме алгоритъм, който да доведе до решение на задачата. Може да се окаже, че реализацията е трудна и да се наложи да търсим друга идея.

Разбиване на задачата на подзадачи

Да се опитаме да дефинираме стъпките, които са ни необходими, за решаването на проблема.

Най-напред, трябва да разделим текста на отделни думи. Това, само по себе си, не е проста стъпка, но е първата ни крачка към разделянето на проблема на по-малки, макар и все още сложни подзадачи.

Следва преброяване на интересуващите ни думи. Това е втората голяма подзадача, която трябва да решим. Да разгледаме двата проблема по отделно и да се опитаме да ги раздробим на още по-прости задачи.

Как да разделим текста на отделни думи?

За да разделим текста на отделни думи, първо трябва да намерим начин да ги идентифицираме. В условието е казано, че за разделител се счита всеки символ, който не е буква. Следователно първо трябва да идентифицираме разделителите и след това да ги използваме за разделянето на текста на думи.

Ето, че се появиха още две подзадачи – намиране на разделителите в текста и разделяне на текста на думи спрямо разделителите. Решения на тези подзадачи можем да реализираме директно.

За намиране на разделителите е достатъчно да обходим всички символи и да извлечем тези, които не са букви.

След като имаме разделителите, можем да реализираме разделянето на текста на думи чрез метода **`split(…)`** на класа **`String`**.

Как да броим думите?

Да предположим, че вече имаме списък с всички думи от текста. Искаме да намерим броя на всички думи, на тези изписани само с главни букви и на тези изписани само с малки букви.

За целта можем да обходим всяка дума от списъка и да проверим дали отговаря на някое от условията, които ни интересуват. На всяка стъпка увеличаваме броя на всички думи. Проверяваме дали текущата дума е изписана само с главни букви и ако това е така увеличаваме броя на думите с главни букви. Аналогично правим проверка и дали думата е изписана само с малки букви.

Така се появяват още две подзадачи – проверка дали дума е изписана само с главни букви и проверка дали е изписана само с малки букви? Те

изглеждат доста лесни. Може би дори е възможно класът **String** да ни предоставя наготово такава функционалност. Проверяваме, но се оказва, че не е така. Все пак забелязваме, че имаме методи, които ни позволяват да преобразуваме символен низ в такъв съставен само от главни или само от малки букви. Това може да ни помогне.

За да проверим дали една дума е съставена само от главни букви е достатъчно да сравним думата с низа, който се получава след като я преобразуваме в дума съставена само от главни букви. Ако са еднакви, значи резултатът от проверката е истина. Аналогична е и проверката за малките букви.

Проверка на идеята

Изглежда, че идеята ни е добра. Разбихме задачата на подзадачи и знаем как да решим всяка една от тях. Дали да не преминем към имплементацията? Пропуснахме ли нещо?

Не трябваше ли да проверим идеята, разписвайки няколко примера на хартия? Може би ще намерим нещо, което сме пропуснали? Можем да започнем с примера от условието:

```
Добре дошли в Софтуерния университет (СОФТУНИ)!
```

Разделителите ще са: интервали**, (,) и !**. За думите получаваме: **Добре, дошли, в, Софтуерния, университет, СОФТУНИ**. Последните два разделителя са един след друг. Какво правим в този случай?

Открихме нещо, за което не бяхме помислили. Изглежда, че когато имаме няколко разделителя един след друг, трябва да ги третираме като един при разделянето на думите.

Преброяваме думите и получаваме коректен резултат. Пробваме още един пример съдържащ няколко последователни разделителя. Ако ги третираме като един, резултатът е коректен.

Изглежда идеята е добра и работи. Можем да пристъпим към имплементацията. За целта ще имплементираме алгоритъма стъпка по стъпка, като на всяка стъпка ще реализираме по една подзадача.

Да помислим за структурите от данни

Задачата е проста и няма нужда от кой знае какви сложни структури от данни.

За разделителите в текста можем да ползваме **String**. При намирането им трябва да ползваме **StringBuilder**, тъй като построяваме низа чрез долепяне на символи.

За думите от текста можем да ползваме масив от низове **String[]** или **ArrayList<String>**.

Да помислим за ефективността

Има ли изисквания за ефективност? Колко най-дълъг може да е текстът? Понеже текстът се въвежда от конзолата, той едва ли ще е много дълъг. Никой няма да въведе 1 MB текст от конзолата. Можем да приемем, че ефективността на решението в случая не е застрашена.

Стъпка 1 – Намиране на разделителите в текста

Ще дефинираме метод, който извлича от текста всички символи, които не са букви и ги връща като символен низ, който след това можем да използваме за разделяне на текста на отделни думи:

```
static String extractSeparators(String text) {
   StringBuilder separators = new StringBuilder();

   int textLength = text.length();
   for (int index = 0; index < textLength; index++) {
      char character = text.charAt(index);

      if (!Character.isLetter(character)) {
         separators.append(character);
      }
   }

   return separators.toString();
}
```

В началото на метода дефинираме обект от класа **StringBuilder**, в който ще натрупваме намерените разделители. Както споменахме в главата "Символни низове", този клас служи за построяване и промяна на символни низове и премахва проблемите с бързодействието при конкатениране на низове.

В цикъл обхождаме всеки един от символите в текста. С помощта на метода **isLetter(…)** на класа **Character**, определяме дали текущия символ е буква и ако не е, го добавяме към разделителите.

Накрая връщаме разделителите като символен низ.

Изпробване на метода extractSeparators(...)

Преди да продължим нататък е редно да изпробваме дали намирането на разделителите работи коректно:

```
public static void main(String[] args) {
   String text = "This is wonderful!!! All separators like " +
         "these ,.(? and these /* are recognized. It works.";
   String separators = extractSeparators(text);
```

```
   System.out.println(separators);
}
```

Стартираме програмата и виждаме дали разделителите са намерени коректно. Резултатът е следният:

```
  !!!     ,.(?   /*  .  .
```

Изпробваме метода и в някои от граничните случаи – текст състоящ се от една дума без разделители, текст съставен само от разделители, празен низ. Изглежда, че методът работи и можем да продължим към реализацията на следващата стъпка.

Стъпка 2 – Разделяна на текста на думи

За разделянето на текста на отделни думи ще използваме разделителите и с помощта на метода **split(…)** на класа **String** ще извършим разделянето.

Преди да подадем разделителите на метода **split(…)** трябва да добавим преди и след тях съответно **[\\Q** и **\\E]+**, понеже методът очаква като параметър регулярен израз.

На езика на регулярните изрази, квадратните скоби следвани от знак +, указват да бъде разпознато всяко срещане на последователност от един или повече от символите заградени в скобите. С **\\Q** и **\\E** указваме, че символите между тях трябва да бъдат третирани като обикновени символи, дори някой от тях да е със специално значение в езика на регулярните изрази. Това е наложително, понеже някой от нашите разделители може да се окаже специален символ, а ние искаме да го третираме като разделител.

Ето как изглежда нашият метод:

```
static String[] extractWords(String text) {
   String separators = extractSeparators(text);

   separators = "[\\Q" + separators + "\\E]+";

   String[] words = text.split(separators);
   return words;
}
```

Преди да преминем към следващата стъпка остава да проверим дали методът работи коректно:

```
public static void main(String[] args) {
   String text =
      "Check it! Separators like $ and ^ should be recognized.";
   String[] words = extractWords(text);
```

```
   for (String word : words) {
      System.out.print(word + " ");
   }
}
```

Резултатът е коректен:

```
Check it Separators like and should be recognized
```

Тук всичко е наред, но когато опитаме да тестваме метода с низ съдържащ само една дума и никакви разделители се хвърля изключение:

```
Exception in thread "main" java.util.regex.PatternSyntaxException:
Unclosed character class near index 2
[\Q\E]+
  ^
   at java.util.regex.Pattern.error(Unknown Source)
   at java.util.regex.Pattern.clazz(Unknown Source)
   at java.util.regex.Pattern.sequence(Unknown Source)
   at java.util.regex.Pattern.expr(Unknown Source)
   at java.util.regex.Pattern.compile(Unknown Source)
   at java.util.regex.Pattern.<init>(Unknown Source)
   at java.util.regex.Pattern.compile(Unknown Source)
   at java.lang.String.split(Unknown Source)
   at java.lang.String.split(Unknown Source)
   at WordsCounter.extractWords(WordsCounter.java:25)
   at WordsCounter.main(WordsCounter.java:82)
```

Оказва се, че когато нямаме разделители, нашият регулярен израз е некоректен. Ако не бяхме тествали метода, грешката щеше да се прояви едва когато някой случайно въведе текст без разделители.

Можем лесно да решим проблема, като веднага след извличането на разделителите проверим дали има такива и ако няма, просто върнем като резултат масив съдържащ един елемент – целият текст:

```
String separators = extractSeparators(text);

if(separators.equals("")){
   String[] words = new String[1];
   words[0] = text;
   return words;
}
```

Проблемът е решен. Тестваме и други гранични случаи като низ съдържащ само разделители и празен низ. Резултатите са коректни.

Стъпка 3 – Определяне дали дума е изписана изцяло с главни или изцяло с малки букви

Вече имаме идея как да имплементираме тези проверки и можем директно да реализираме методите:

```
static boolean isUpperCase(String word) {
   boolean result = word.equals(word.toUpperCase());
   return result;
}

static boolean isLowerCase(String word) {
   boolean result = word.equals(word.toLowerCase());
   return result;
}
```

Изпробваме ги подавайки им думи съдържащи само главни, само малки и такива съдържащи главни и малки букви. Резултатите са коректни.

Стъпка 4 – Преброяване на думите

Вече можем да пристъпим към решаването на проблема – преброяването на думите. Трябва само да обходим списъка с думите и в зависимост каква е думата да увеличим съответните броячи, след което да отпечатаме резултата:

```
static void countWords(String[] words) {
   int totalCount = 0;
   int allUpperCaseCount = 0;
   int allLowerCaseCount = 0;

   for (String word : words) {
      totalCount++;
      if (isUpperCase(word)) {
         allUpperCaseCount++;
      } else if (isLowerCase(word)) {
         allLowerCaseCount++;
      }
   }

   System.out.printf("Total words count: %s\n", totalCount);
   System.out.printf("Upper case words count: %s\n",
         allUpperCaseCount);
   System.out.printf("Lower case words count: %s\n",
         allLowerCaseCount);
    }
```

Нека проверим дали броенето работи коректно:

```
public static void main(String[] args) {
   String[] words = {"This", "is", "our", "TEST", "case"};
   countWords(words);
}
```

Стартираме приложението и получаваме верен резултат:

```
Total words count: 5
Upper case words count: 1
Lower case words count: 3
```

Проверяваме резултатите и в граничните случаи, когато списъкът съдържа думи само с главни или само с малки букви, както и когато списъкът е празен.

Стъпка 5 – Вход от конзолата

Остава да реализираме и последната стъпка, даваща възможност на потребителя да въвежда текст:

```
static String readText() {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter text:");
   String text = input.nextLine();

   return text;
}
```

Проверката е лесна. Трябва само да въведем текст, след което да го отпечатаме в конзолата и да се уверим, че двата текста са еднакви:

```
public static void main(String[] args) {
   String text = readText();
   System.out.println(text);
}
```

Резултатът е коректен:

```
Enter text:
 This is our text.
 This is our text.
```

Стъпка 6 – Сглобяване на всички части в едно цяло

След като сме решили всички подзадачи, можем да пристъпим към пълното решаване на проблема. Остава да добавим **main(…)** метод, в който да съединим отделните парчета:

```
public static void main(String[] args) {
   String text = readText();
   String[] words = extractWords(text);
   countWords(words);
}
```

Тестване на решението

Макар внимателно да сме тествали решението на всяка подзадача, все още ни предстои тестване на цялостното решение. Трябва да направим това, за да проверим дали отделните части си пасват добре.

Започваме от общия случай, подавайки примерен текст, съдържаш последователност от думи, измежду които такива само с главни и само с малки букви, както и последователност от няколко разделителя един след друг:

```
public static void main(String[] args) {
   String text = "We need several separators " +
         "like ! , ? and UPPER CASE words " +
         "and lower case words. This is all.";
   String[] words = extractWords(text);
   countWords(words);
}
```

Изпълняваме програмата и се уверяваме, че резултатът е коректен:

```
Total words count: 16
Upper case words count: 2
Lower case words count: 12
```

Изпробваме с още няколко подобни примера и след като сме сигурни, че всичко е наред в този случай, преминаваме към тестването на граничните случаи. Проверяваме какви са резултатите, когато текста съдържа само думи с главни букви или само думи с малки букви. Пробваме да въведем само разделители или само една дума без разделители. Тестваме и случай, когато текстът е празен низ.

Оказва се, че във всички случай, с изключение на този, в който подаваме празен низ, резултатите са коректни. Трябва да намерим от къде идва грешката.

Оказва се, че когато подадем празен низ, резултатния списък, получен след разделянето му на думи, съдържа един елемент, който е празен низ. Когато броим думите, ние го броим като дума с главни букви, защото при преобразуването му към горен регистър и сравнението двата низа съвпадат.

Можем да решим проблема по различни начини. Възможно решение е, при разделянето на думите да връщаме празен списък, ако низът е празен. Друг вариант е директно да извеждаме резултата, без да броим, ако въведеният низ е празен. Трети вариант е да не броим думите, състоящи се от празен низ.

Понеже грешката идва от неправилно броене на празния низ като дума, можем да се спрем на третия вариант и да направим съответните промени в кода за броене на думите. Когато срещнем дума състояща се от празен низ, няма да я броим:

```
for (String word : words) {
   if (word.equals("")) {
      continue;
   }

   totalCount++;
   if (isUpperCase(word)) {
      allUpperCaseCount++;
   } else if (isLowerCase(word)) {
      allLowerCaseCount++;
   }
}
```

Вече програмата ни работи и в този случай. Повтаряме отново и другите тестове, за да се уверим, че с оправянето на този проблем не сме създали други. Оказва се, че всичко е наред. Можем да направим тест и с по-голям текст от няколко страници, за де се уверим, че нямаме проблем с бързодействието.

Ето как изглежда кодът на цялостното решение след приложените корекции:

WordsCounter.java

```
import java.util.Scanner;

public class WordsCounter {

   static String extractSeparators(String text) {
      StringBuilder separators = new StringBuilder();

      int textLength = text.length();
      for (int index = 0; index < textLength; index++) {
         char character = text.charAt(index);

         if (!Character.isLetter(character)) {
            separators.append(character);
```

```
      }
   }

   return separators.toString();
}

static String[] extractWords(String text) {
   String separators = extractSeparators(text);

   if (separators.equals("")) {
      String[] words = new String[1];
      words[0] = text;
      return words;
   }

   separators = "[\\Q" + separators + "\\E]+";

   String[] words = text.split(separators);
   return words;
}

static boolean isUpperCase(String word) {
   boolean result = word.equals(word.toUpperCase());
   return result;
}

static boolean isLowerCase(String word) {
   boolean result = word.equals(word.toLowerCase());
   return result;
}

static void countWords(String[] words) {
   int totalCount = 0;
   int allUpperCaseCount = 0;
   int allLowerCaseCount = 0;

   for (String word : words) {
      if (word.equals("")) {
         continue;
      }

      totalCount++;
      if (isUpperCase(word)) {
         allUpperCaseCount++;
      } else if (isLowerCase(word)) {
         allLowerCaseCount++;
      }
   }
```

```
        System.out.printf("Total words count: %s\n", totalCount);
        System.out.printf("Upper case words count: %s\n",
              allUpperCaseCount);
        System.out.printf("Lower case words count: %s\n",
              allLowerCaseCount);
    }

    static String readText() {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter text:");
        String text = input.nextLine();

        return text;
    }

    public static void main(String[] args) {
        String text = readText();
        String[] words = extractWords(text);
        countWords(words);
    }
}
```

Дискусия за производителността

Тъй като въпросът за производителността в тази задача не е явно поставен, само ще дадем идея как бихме могли да реагираме, ако евентуално се окаже, че нашият алгоритъм е бавен. Понеже разделянето по регулярен израз предполага, че целият текст трябва да бъде прочетен в паметта и думите, получени при разделянето също трябва да се запишат в паметта, то програмата ще консумира голямо количество памет, ако входният текст е голям. Например, ако входът е 200 MB текст, програмата ще изразходва най-малко 800 MB памет, понеже всяка дума се пази 2 пъти по 2 байта за всеки символ.

Ако искаме да избегнем консумацията на голямо количество памет, трябва да не пазим всички думи едновременно в паметта. Можем да измислим друг алгоритъм: сканираме текста символ по символ и натрупваме буквите в някакъв буфер (примерно **`StringBuilder`**). Ако срещнем в даден момент разделител, то в буфера би трябвало да стои поредната дума. Можем да я анализираме дали е с малки или главни букви и да зачистим буфера. Това можем да повтаряме до достигане на края на файла. Изглежда по-ефективно, нали?

За по-ефективно проверяване за главни/малки букви можем да направим цикъл по буквите и проверка на всяка буква. Така ще си спестим преобразуването в горен/долен регистър, което заделя излишно памет за всяка

проверена дума, която след това се освобождава и в крайна сметка това отнема процесорно време.

Очевидно второто решение е по-ефективно. Възниква въпросът трябва ли след като сме написали първото решение да го изхвърлим и да напишем съвсем друго решение. Всичко зависи от изискванията за ефективност. В условието на задачата няма предпоставки да смятаме, че ще ни подадат като вход стотици мегабайти. Следователно решението с регулярните изрази също е коректно и ще ни свърши работа.

Задача 2: Матрица с прости числа

Напишете програма, която прочита от стандартния вход цяло положително число N и отпечатва първите N^2 прости числа в квадратна матрица с размери N x N. Запълването на матрицата трябва да става по редове от първия към последния и отляво надясно.

Забележка: Едно естествено число наричаме просто, ако няма други делители освен 1 и себе си. Числото 1 не се счита за просто.

Примерен вход:

2	**3**	**4**

Примерен изход:

2 3 **5 7**	**2 3 5** **7 11 13** **17 19 23**	**2 3 5 7** **11 13 17 19** **23 29 31 37** **41 43 47 53**

Намиране на подходяща идея за решение

Можем да решим задачата, като с помощта на два вложени цикъла отпечатаме редовете и колоните на резултатната матрица. За всеки неин елемент ще извличаме и отпечатваме поредното просто число.

Разбиване на задачата на подзадачи

Трябва да решим поне две подзадачи – намиране на поредното просто число и отпечатване на матрицата. Отпечатването на матрицата можем да направим директно, но за намирането на поредното просто число, ще трябва да помислим малко. Може би най-интуитивният начин, който ни идва наум за това е започвайки от предходното намерено просто число, да проверяваме всяко следващо дали е просто и в момента, в който това се окаже истина, да го върнем като резултат. Така на хоризонта се появява още една подзадача – проверка дали дадено число е просто.

Проверка на идеята

Нашата идея за решение на задачата директно получава търсения в условието резултат. Разписваме 1-2 примера на хартия и се убеждаваме, че работи.

Да помислим за структурите от данни

В тази задача се ползва една единствена структура от данни – матрицата. Естествено е да ползваме двумерен масив.

Да помислим за ефективността

Понеже изходът е на конзолата, при особено големи матрици (примерно 1000 x 1000) резултатът няма да може да се визуализира добре. Това означава, че задачата трябва да се реши за разумно големи матрици, но не прекалено големи, примерно за N ≤ 200. При нашия алгоритъм при N=200 ще трябва да намерим първите 40 000 прости числа, което не би трябвало да е бавно.

Стъпка 1 – Проверка дали дадено число е просто

За проверката дали дадено число е просто, можем да дефинираме метод **`isPrime(…)`**. За целта е достатъчно да проверим, че то не се дели без остатък на никое от предхождащите го числа. За да сме още по-точни, достатъчно е да проверим, че то не се дели на никое от числата между 2 и корен квадратен от числото. Това е така, защото, ако числото **`p`** има делител **`x`**, то **`p = x.y`** и поне едно от числата **`x`** и **`y`** ще е по-малко или равно на корен квадратен от **`p`**. Следва реализация на метода:

```
static boolean isPrime(int number) {
   int maxDivider = (int) Math.sqrt(number);

   for (int divider = 2; divider <= maxDivider; divider++) {
      if (number % divider == 0) {
         return false;
      }
   }
   return true;
}
```

Можем да се уверим, че методът работи коректно, подавайки му последователно различни числа, някои от които прости, и проверявайки върнатия резултат.

Стъпка 2 – Намиране на следващото просто число

За намирането на следващото просто число можем да дефинираме метод, който приема като параметър число, и връща като резултат първото, по-голямо от него, просто число. За проверката дали число е просто, ще използваме методът от предишната стъпка. Следва реализацията на метода:

```
static int findNextPrime(int startNumber) {
   int number = startNumber;
   while (!isPrime(number)) {
      number++;
   }
   return number;
}
```

Отново трябва да изпробваме метода подавайки му няколко числа и проверявайки, дали резултатът е правилен.

Стъпка 3 – Отпечатване на матрицата

След като дефинирахме горните методи, вече сме готови да отпечатаме и цялата матрица:

```
static void printMatrix(int dimension){
   int lastPrime = 1;
   for (int row = 0; row < dimension; row++) {
      for (int col = 0; col < dimension; col++) {
         int nextPrime = findNextPrime(lastPrime + 1);
         System.out.printf(" %d", nextPrime);
         lastPrime = nextPrime;
      }
      System.out.println();
   }
}
```

Стъпка 4 – Вход от конзолата

Остава да добавим възможност за прочитане на N от конзолата:

```
static int readN() {
   Scanner input = new Scanner(System.in);
   System.out.print("N = ");
   int n = input.nextInt();
   return n;
}
```

```
public static void main(String[] args) {
   int n = readN();
   printMatrix(n);
}
```

Тестване на решението

След като всичко е готово, можем да пристъпим към проверка на решението. За целта можем да намерим примерно първите 25 прости числа и да проверим изхода на програмата за стойности на N от 1 до 5. Не трябва да пропускаме случая за N=1, понеже това е граничен случай и вероятността за допусната грешка при него е значително по-голяма.

В конкретния случай, при условие че сме тествали добре методите на всяка стъпка, можем да се задоволим и с примерите от условието на задачата. Ето как изглежда изходът от програмата за стойности на N съответно 1, 2, 3 и 4:

```
2          2 3                2 3 5                2 3 5 7
           5 7                7 11 13                 11 13 17 19
                              17 19 23                23 29 31 37
                                                      41 43 47 53
```

Можем да се уверим, че решението на задачата работи сравнително бързо и за по-големи стойности на N. Примерно при N=200 не се усеща някакво забавяне.

Следва пълната реализация на решението:

PrimesMatrix.java

```
import java.util.Scanner;

public class PrimesMatrix {

   static boolean isPrime(int number) {
      int maxDivider = (int) Math.sqrt(number);

      for (int divider = 2; divider <= maxDivider; divider++) {
         if (number % divider == 0) {
            return false;
         }
      }
      return true;
   }

   static int findNextPrime(int startNumber) {
```

```
      int number = startNumber;
      while (!isPrime(number)) {
         number++;
      }
      return number;
   }

   static void printMatrix(int dimension) {
      int lastPrime = 1;
      for (int row = 0; row < dimension; row++) {
         for (int col = 0; col < dimension; col++) {
            int nextPrime = findNextPrime(lastPrime + 1);
            System.out.printf(" %d", nextPrime);
            lastPrime = nextPrime;
         }
         System.out.println();
      }
   }

   static int readN() {
      Scanner input = new Scanner(System.in);
      System.out.print("N = ");
      int n = input.nextInt();
      return n;
   }

   public static void main(String[] args) {
      int n = readN();
      printMatrix(n);
   }
}
```

Дискусия за производителността

Трябва да отбележим, че посоченото решение не търси простите числа по най-ефективния начин. Въпреки това, с оглед яснотата на изложението и поради очаквания малък размер на матрицата, можем да използваме този алгоритъм, без да имаме проблеми с производителността.

Ако трябва да подобрим производителността, можем да намерим първите N^2 числа с "решето на Ератостен" (sieve of Eratosthenes) без да проверяваме дали всяко число е просто до намиране на N^2 прости числа.

Задача 3: Аритметичен израз

Напишете програма, която изчислява стойността на прост аритметичен израз, съставен от цели числа без знак и аритметичните операции "+" и "-". Между числата няма интервали.

Изразът се задава във формат:

```
<число><операция>...<число>
```

Примерен вход:

```
1+2-7+2-1+28+2+3-37+22
```

Примерен изход:

```
15
```

Намиране на подходяща идея за решение

За решаване на задачата можем да използваме факта, че формата на израза е стриктен и ни гарантира, че имаме последователност от число, операция, отново число и т.н.

Така можем да извлечем всички числа участващи в израза, след това всички оператори и накрая да изчислим стойността на израза, комбинирайки числата с операторите.

Проверка на идеята

Наистина, ако вземем лист и химикал и изпробваме подхода с няколко израза, получаваме верен резултат. Първоначално резултатът е равен на първото число, а на всяка следващата стъпка добавяме или изваждаме следващото число в зависимост от текущия оператор.

Структури от данни и ефективност

Задачата е прекалено проста, за да ползваме сложни структури от данни. Числата и значите можем да запишем в масив или `ArrayList`. За проблеми с ефективността не можем да говорим, тъй като всеки знак и всяко число се обработват точно по веднъж, т.е. имаме линейна сложност на алгоритъма.

Разбиване на задачата на подзадачи

След като сме се убедили, че идеята работи можем да пристъпим към разбиването на задачата на подзадачи. Първата подзадача, която ще трябва да решим е извличането на числата от израза. Втората ще е извличането на операторите. Накрая ще трябва да изчислим стойността на целия израз, използвайки числата и операторите, които сме намерили.

Стъпка 1 – Извличане на числата

За извличане на числата е необходимо да разделим израза, като за разделители използваме операторите. Това можем да направим лесно чрез метода **split(…)** на класа **String**. След това ще трябва да преобразуваме получения масив от символни низове в масив от цели числа:

```
static int[] extractNumbers(String expression) {
   String[] splitResult = expression.split("[+-]");

   int numbersCount = splitResult.length;
   int[] numbers = new int[numbersCount];

   int currentNumber;
   for (int index = 0; index < numbersCount; index++) {
      currentNumber = Integer.parseInt(splitResult[index]);
      numbers[index] = currentNumber;
   }

   return numbers;
}
```

За преобразуването на символните низове в цели числа използваме метода **parseInt(…)** на класа **Integer**. Той приема като параметър символен низ и връща като резултат целочислената стойност, представена от него.

Защо използваме масив за съхранение на числата? Не можем ли да използваме свързан списък или **ArrayList**? Разбира се, че можем, но в случая е нужно единствено да съхрани числата и след това да ги обходим при изчисляването на резултата. Ето защо масивът ни е напълно достатъчен.

Преди да преминем към следващата стъпка проверяваме дали извличането на числата работи коректно:

```
public static void main(String[] args) {
   String expression = "1+2-7+2-1+28";
   int[] numbers = extractNumbers(expression);
   for (int number : numbers) {
      System.out.printf("%s ", number);
   }
}
```

Резултатът е точно такъв, какъвто трябва:

```
1 2 7 2 1 28
```

Проверяваме и граничния случай, когато изразът се състои само от едно число без оператори и се уверяваме, че и той се обработва добре.

Стъпка 2 – Извличане на операторите

Извличането на операторите може да направим аналогично на извличането на числата, но като вземем предвид, че разделител може да е последователност от няколко цифри:

```
static String[] extractOperators(String expression) {
   String[] operators = expression.split("[0123456789]+");
   return operators;
}
```

Следва проверка, дали методът работи коректно:

```
public static void main(String[] args) {
   String expression = "1+2-7+2-1+28";
   String[] operators = extractOperators(expression);
   for (String operator : operators) {
      System.out.printf("'%s' ", operator);
   }
}
```

Забелязваме, че в резултата получаваме един празен низ, който е излишен:

```
'' '+' '-' '+' '-' '+'
```

Това е така, понеже на първа позиция винаги стои цифра и разделяйки низа спрямо последователност от цифри, получаваме празния низ в началото. Можем да решим лесно проблема, като премахнем първия елемент от списъка, преди да върнем резултата:

```
static String[] extractOperators(String expression) {
   String[] operators = expression.split("[0123456789]+");

   int operatorsCount = operators.length;
   if (operatorsCount > 0) {
      operators = Arrays.copyOfRange(operators,1,operatorsCount);
   }
   return operators;
}
```

Проверяваме отново резултата и този път е коректен. Правим проверка и за граничния случай, когато изразът не съдържа оператори, а се състои само от едно число. В този случай получаваме празен низ, което е очакваното поведение.

Стъпка 3 – Изчисляване на стойността на израза

За изчисляване на стойността на израза, можем да използваме факта, че числата винаги са с едно повече от операторите и с помощта на един цикъл да изчислим стойността на израза, при условие че са ни дадени списъците с числата и операторите:

```
static int calculateExpression(int[] numbers,String[] operators) {
   int result = numbers[0];

   for (int i = 1; i < numbers.length; i++) {
      String nextOperator = operators[i - 1];
      int nextNumber = numbers[i];

      if (nextOperator.equals("+")) {
         result += nextNumber;
      } else if (nextOperator.equals("-")) {
         result -= nextNumber;
      }
   }
   return result;
}
```

Проверяваме работата на метода:

```
public static void main(String[] args) {
   // Expression: 1 + 2 - 3 + 4
   int[] numbers = { 1, 2, 3, 4 };
   String[] operators = { "+", "-", "+" };

   // Expected result: 4
   int result = calculateExpression(numbers, operators);
   System.out.printf("Result is: %s", result);
}
```

Резултатът е коректен:

```
Result is: 4
```

Стъпка 4 – Вход от конзолата

Ще трябва да дадем възможност на потребителя да въвежда израз:

```
static String readExpression() {
   Scanner input = new Scanner(System.in);
   System.out.print("Enter expression: ");
   String expression = input.nextLine();
```

```
    return expression;
}
```

Стъпка 5 – Сглобяване на всички части в едно цяло

Остава ни само да накараме всичко да работи заедно:

```
public static void main(String[] args) {
    String expression = readExpression();

    int[] numbers = extractNumbers(expression);
    String[] operators = extractOperators(expression);

    int result = calculateExpression(numbers, operators);
    System.out.printf("%s = %d \n", expression, result);
}
```

Тестване на решението

Можем да използваме примера от условието на задачата, при тестването на решението. Получаваме коректен резултат:

```
Enter expression: 1+2-7+2-1+28+2+3-37+22
1+2-7+2-1+28+2+3-37+22 = 15
```

Трябва да направим още няколко теста с различни примери, включително и случаят, когато изразът се състои само от едно число, за да се уверим, че решението ни работи.

Можем да тестваме и празен низ. Не е много ясно това дали е коректен вход, но може да го предвидим за всеки случай. Освен това не е ясно какво става, ако някой въведе интервали в израза, примерно вместо "**2+3**" въведе "**2 + 3**". Хубаво е да предвидим тези ситуации.

Друго, което забравихме да тестваме, е какво става при число, която не се събира в типа **int**. Какво ще стане, ако ни бъде подаден изразът "**11111111111111111111111111111+22222222222222222222222222222**"?

Дребни поправки и повторно тестване

Във всички случаи, когато изразът е невалиден, ще се получи някакво изключение (най-вероятно **NumberFormatException**). Достатъчно е да прихванем изключенията и при настъпване на изключение да съобщим, че е въведен грешен израз. Следва пълната реализация на решението след тази корекция:

SimpleExpressionEvaluator.java

```
import java.util.Arrays;
import java.util.Scanner;

public class SimpleExpressionEvaluator {

   public static void main(String[] args) {
      String expression = readExpression();
      try {
         int[] numbers = extractNumbers(expression);
         String[] operators = extractOperators(expression);

         int result = calculateExpression(numbers, operators);
         System.out.printf("%s = %d \n", expression, result);
      } catch (Exception ex) {
         System.out.println("Invalid expression!");
      }
   }

   private static int[] extractNumbers(String expression) {
      String[] splitResult = expression.split("[+-]");

      int numbersCount = splitResult.length;
      int[] numbers = new int[numbersCount];

      int currentNumber;
      for (int index = 0; index < numbersCount; index++) {
         currentNumber = Integer.parseInt(splitResult[index]);
         numbers[index] = currentNumber;
      }

      return numbers;
   }

   private static String[] extractOperators(String expression) {
      String[] operators = expression.split("[0123456789]+");

      int operatorsCount = operators.length;
      if (operatorsCount > 0) {
         operators=Arrays.copyOfRange(operators,1,operatorsCount);
      }
      return operators;
   }

   private static int calculateExpression(int[] numbers,
         String[] operators) {
```

```
      int result = numbers[0];

      for (int i = 1; i < numbers.length; i++) {
         String nextOperator = operators[i - 1];
         int nextNumber = numbers[i];

         if (nextOperator.equals("+")) {
            result += nextNumber;
         } else if (nextOperator.equals("-")) {
            result -= nextNumber;
         }
      }
      return result;
   }

   private static String readExpression() {
      Scanner input = new Scanner(System.in);
      System.out.print("Enter expression: ");
      String expression = input.nextLine();

      return expression;
   }
}
```

Упражнения

1. Решете задачата "броене на думи в текст", без да ползвате регулярни изрази, само с един буфер (**StringBuilder**).
2. Реализирайте по-ефективно решение на задачата "матрица с прости числа" като търсите простите числа с "решето на Ератостен": http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.
3. Добавете поддръжка на операциите умножение и целочислено деление в задачата "аритметичен израз". Имайте предвид, че те са с по-висок приоритет от събирането и изваждането!
4. Добавете поддръжка на реални числа, не само цели.
5. Направете възможни пресмятанията с числа, които не се събират в стандартните типове **float** и **double**, примерно числа със 100 цифри преди и 200 цифри след десетичната запетая.
6. Добавете поддръжка на скоби в задачата "аритметичен израз".
7. Напишете програма, която валидира аритметичен израз. Например "2*(2.25+5.25)-17/3" е валиден израз, докато "*232*-25+(33+a" е невалиден.

Решения и упътвания

1. Можете да четете входния файл символ по символ. Ако поредният символ е буква, го добавяте към буфера, а ако е разделител, анализирате буфера (той съдържа поредната дума) и след това зачиствате буфера. Когато свърши входния файл, трябва да анализирате последната дума, която е в буфера (ако файлът не завършва с разделител). Изглежда лесно и просто.

2. Помислете първо колко прости числа ви трябват. След това помислете до каква стойност трябва да пускате решето на Ератостен, за да ви стигнат простите числа за запълване на матрицата. Можете опитно да измислите някаква формула.

3. Достатъчно е да изпълните първо всички умножения и деления, а след тях всички събирания. Помислихте ли за деление на нула?

4. Работата с реални числа можете да осигурите като разрешите използването на символа "." и заместите **`int`** с **`double`**.

5. Разгледайте как работи класа **`BigDecimal`** и нанесете необходимите корекции, за да използвате този клас вместо типа **`double`** за съхранение на числата и междинните резултати при пресмятането на израза.

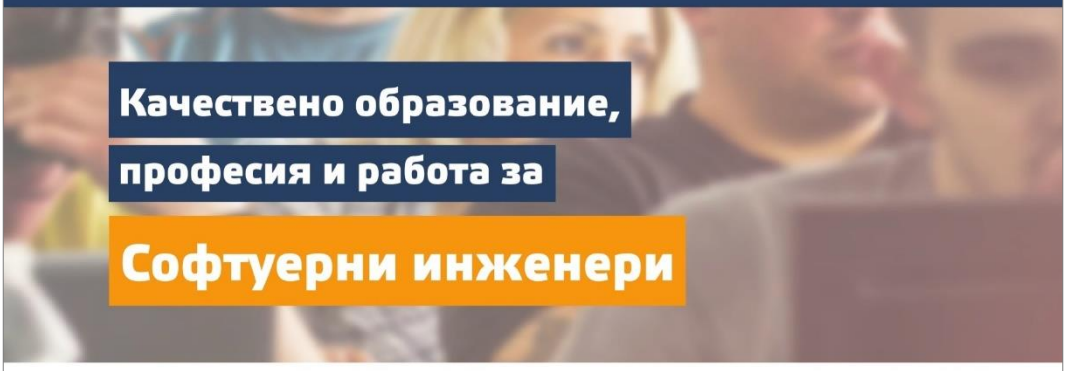
6. Можете да направите така: намирате първата затваряща скоба и търсите наляво съответната ѝ отваряща скоба. Това, което е в скобите е аритметичен израз без скоби, за който вече имаме алгоритъм за изчисление на стойността му. Можем да го заместим със стойността му. Повтаряме това за следващите скоби докато скобите свършат. Накрая ще имаме израз без скоби.

   Например, ако имаме "2*((3+5)*(4-7*2))", ще заместим "(3+5)" с 8, след това "(4-7*2)" с -10. Накрая ще заместим (8*-10) с -80 и ще сметнем 2*-80, за да получим резултата -160. Трябва да предвидим аритметични операции с отрицателни числа, т.е. да позволяваме числата да имат знак.

   Другият алгоритъм много по-лесен. Използва се стек и преобразуване на израза до "обратен полски запис". Можете да потърсите в Интернет за фразата "**`postfix notation`**" и за "**`shunting yard" algorithm`**.

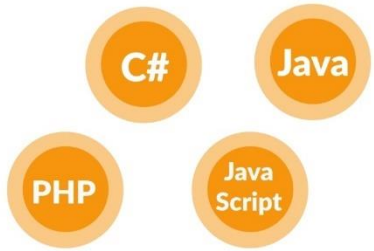
7. Ако изчислявате израза с обратен полски запис, можете да допълните алгоритъма, така че да проверява за валидност на израза. Добавете следните правила: когато очаквате число, а се появи нещо друго, изразът е невалиден. Когато очаквате аритметична операция, а се появи нещо друго, изразът е невалиден. Когато скобите не си съответстват, ще препълните стека или ще останете накрая с недоизпразнен стек. Помислете за специални случаи, примерно "**`-1`**", "**`-(2+4)`**" и др.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Глава 25. Примерен изпит по програмиране – 11.12.2005 г.

Автор

Теодор Стоев

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на няколко примерни задачи от изпит в НАРС, проведен на 11.12.2005 г. При решаването на задачите ще се придържаме към съветите от главата "Как да решаваме задачи по програмиране".

Задача 1: Квадратна матрица

По дадено число N (въвежда се от клавиатурата) да се генерира и отпечата квадратна матрица, съдържаща числата от 0 до N^2-1, разположени като спирала, започваща от центъра на матрицата и движеща се по часовниковата стрелка, тръгвайки в началото надолу (вж. примерите).

Примерен резултат при N=3 и N=4:

4	5	6
3	0	7
2	1	8

15	4	5	6
14	3	0	7
13	2	1	8
12	11	10	9

Решение на задачата

От условието лесно се вижда, че имаме поставена алгоритмична задача (макар и с неголяма сложност). Това е и основната част от решението на задачата – да измислим подходящ алгоритъм за запълване на клетките на квадратна матрица по описания начин. Ще покажем на читателя типичните разсъждения необходими за решаването на този конкретен проблем.

Да започнем с избора на структура от данни за представяне на матрицата. Удобно е да имаме директен достъп до всеки елемент на матрицата, затова

ще се спрем на двумерен масив **matrix** от целочислен тип. При стартирането на програмата прочитаме от стандартния вход размерността **n** на матрицата и я инициализираме по следния начин:

```
int[][] matrix = new int[n][n];
```

Измисляне на идея за решение

Време е да измислим идеята на алгоритъма, който ще имплементираме. Трябва да запълним матрицата с числата от 0 до N^2-1 и веднага съобразяваме, че това може да стане с помощта на цикъл, който на всяка итерация поставя едно от числата в предназначената за него клетка на матрицата. Текущата позиция ще представяме чрез целочислените променливи **positionX** и **positionY** – двете координати на позицията. Да приемем, че знаем началната позиция – тази, на която трябва да поставим първото число. По този начин задачата се свежда до намиране на метод за определяне на всяка следваща позиция, на която трябва да бъде поставено число – това е нашата главна подзадача.

Подходът за определяне на следващата позиция спрямо текущата е следният: търсим строга закономерност при спираловидното движение по клетките. Започваме от най-очевидното нещо – движението винаги е по посока на часовниковата стрелка, като първоначално посоката е надолу. Дефинираме целочислена променлива **direction**, която ще показва текущата посока на движение. Тази променлива ще приема стойностите 0 (надолу), 1 (наляво), 2 (нагоре) и 3 (надясно). При смяна на посоката на движение просто увеличаваме с единица стойността на direction и делим по модул 4 (за да получаваме само стойности от 0 до 3).

Следващата стъпка при съставянето на алгоритъма е да установим кога се сменя посоката на движение (през колко итерации на цикъла). От двата примера можем да забележим, че броят на итерациите, през които се сменя посоката образува нестрого растящите редици 1, 1, 2, 2, 2 и 1, 1, 2, 2, 3, 3, 3. Ако разпишем на лист хартия по-голяма матрица от същия вид ясно виждаме, че редицата на смените на посоката следва същата схема – числата през едно нарастват с 1, като последното число не нараства. За моделирането на това поведение ще използваме променливите **stepsCount** (броят на итерациите в текущата посока), **stepPosition** (номерът на поредната итерация в тази посока) и **stepChange** (флаг, показващ дали на текущата итерация трябва да увеличим стойността на **stepCount**).

Проверка на идеята

Сега, нека проверим идеята. Пробваме на N=0. Разписваме алгоритъма набързо на лист хартия. Изглежда работи. Пробваме за N=1. Работи. Пробваме за N=2. Работи. Пробваме за N=3. Работи. Стига толкова сме проверявали. Можем да преминем към имплементация.

Структури от данни и ефективност

При тази задачата за структурите от данни нямаме много избор. Матрицата ще пазим в двумерен масив. Други данни нямаме (освен числа). С ефективността няма да имаме проблем, тъй като програмата ще направи толкова стъпки, колкото са елементите в матрицата, т.е. имаме линейна сложност.

Реализация на идеята: стъпка по стъпка

Нека видим как можем да реализираме тази идея като код:

```
for (int i = 0; i < count; i++) {
   matrix[positionY][positionX] = i;

   if (stepPosition < stepsCount) {
      stepPosition++;
   else {
      stepPosition = 1;

      if (stepChange == 1) {
         stepsCount++;
      }
      stepChange = (stepChange + 1) % 2;

      direction = (direction + 1) % 4;
   }

   switch (direction) {
      case 0: positionY++; break;
      case 1: positionX--; break;
      case 2: positionY--; break;
      case 3: positionX++; break;
   }
}
```

Тук е моментът да отбележим, че е голяма рядкост да съставим тялото на подобен цикъл от първия път, без да сгрешим. Вече знаем за правилото да пишем кода стъпка по стъпка, но за тялото на този цикъл то е трудно приложимо – нямаме ясно обособени подзадачи, които можем да тестваме независимо една от друга. Това не бива да ни притеснява – можем да използваме мощния debugger на Eclipse за постъпково проследяване на изпълнението на кода. По този начин лесно ще открием къде е грешката, ако има такава.

След като имаме добре измислена идея на алгоритъм (дори да не сме напълно сигурни, че така написаният код работи безпроблемно), остава да дадем начални стойности на вече дефинираните променливи и да отпечатаме получената след изпълнението на цикъла матрица.

Ясно е, че броят на итерациите на цикъла е точно N^2 и затова инициализираме променливата **count** с тази стойност. От двата дадени примера и нашите собствени (написани на лист) примери определяме началната позиция в матрицата в зависимост от четността на нейната размерност:

```
int positionX = n / 2;
int positionY = n % 2 == 0 ? n / 2 - 1 : n / 2;
```

На останалите променливи даваме еднозначно следните стойности (вече обяснихме каква е тяхната семантика):

```
int direction = 0;
int stepsCount = 1;
int stepPosition = 0;
int stepChange = 0;
```

Последната подзадача, която трябва да решим, за да имаме работеща програма, е отпечатването на матрицата на стандартния изход. Това става най-лесно с два вложени цикъла, които я обхождат по редове и на всяка итерация на вътрешния цикъл прилагаме подходящото форматиране:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      System.out.printf("%3d ", matrix[i][j]);
   }
   System.out.println();
}
```

С това изчерпахме основните съставни елементи на програмата. Следва пълният изходен код на нашето решение:

MatrixSpiral.java

```
import java.util.*;

public class MatrixSpiral {
   public static void printMatrix(int[][] matrix, int n) {
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            System.out.printf("%3d ", matrix[i][j]);
         }
         System.out.println();
      }
   }

   public static void fillMatrix(int[][] matrix, int n) {
      int count = n * n;
```

```
      int positionX = n / 2;
      int positionY = n % 2 == 0 ? n / 2 - 1 : n / 2;
      int direction = 0;
      int stepsCount = 1;
      int stepPosition = 0;
      int stepChange = 0;

      for (int i = 0; i < count; i++) {
         matrix[positionY][positionX] = i;

         if (stepPosition < stepsCount) {
            stepPosition++;
         } else {
            stepPosition = 1;

            if(stepChange == 1) {
               stepsCount++;
            }
            stepChange = (stepChange + 1) % 2;

            direction = (direction + 1) % 4;
         }

         switch (direction) {
            case 0: positionY++; break;
            case 1: positionX--; break;
            case 2: positionY--; break;
            case 3: positionX++; break;
         }
      }
   }

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("N = ");
      int n = input.nextInt();

      int[][] matrix = new int[n][n];

      fillMatrix(matrix, n);

      printMatrix(matrix, n);
   }
}
```

Тестване на решението

След като сме имплементирали решението, уместно е да го тестваме с достатъчен брой стойности на N, за да се уверим, че работи правилно. Започваме с примерните стойности 3 и 4, а после проверяваме и за 5, 6, 7, 8, 9, … Важно е да тестваме и за граничните случаи: 0 и 1. Провеждаме необходимите тестове и се убеждаваме, че всичко работи. В случая не е уместно да тестваме за скорост (примерно с N=1000), защото при голямо N изходът е прекалено обемен и задачата няма особен смисъл.

Задача 2: Броене на думи в текстов файл

Даден е текстов файл **words.txt**, който съдържа няколко думи, по една на ред. Да се напише програма, която намира броя срещания на всяка от дадените думи като подниз във файла **sample.txt**. Главните и малките букви се считат за еднакви. Резултатът да се запише в текстов файл с име **result.txt** във формат **<дума> - <брой срещания>**.

Примерен входен файл **words.txt**:

```
for
academy
student
develop
```

Примерен входен файл **sample.txt**:

```
The National Academy for Software Development is a center for
professional training for software engineers. The Academy offers
courses designed to develop practical computer programming skills.
Students finished the Academy are guaranteed to have a job as a software
developers.
```

Примерен резултатен файл **result.txt**:

```
for – 3
academy – 3
student – 1
develop – 3
```

Решение на задачата

В дадената задача акцентът е не толкова върху алгоритъма за нейното решаването, а по-скоро върху техническата реализация. За да напишем решението, трябва да сме добре запознати с работата с файлове в Java, както и с основните структури от данни.

Измисляне на идея за решение

При тази задача идеята за решение е очевидна: прочитаме файла с думите, след това минаваме през текста и за всяка дума в него проверяваме дали е от интересните за нас думи и увеличаваме съответния брояч. Няма нищо трудно за измисляне.

Проверка на идеята

Идеята за решаване е тривиална, но все пак можем да я проверим като разпишем на лист хартия какво ще се получи за примерния входен файл. Лесно се убеждаваме, че тази идея е правилна.

Разделяме задачата на подзадачи

При реализацията на програмата можем да отделим три основни стъпки (подзадачи):

1. Прочитаме файла **words.txt** и добавяме всяка дума от него към списък **words** (за целта използваме **ArrayList** реализацията). За четенето на текстови файлове е удобно да използваме класа **Scanner**, който вече подробно сме разглеждали.
2. Обхождаме в цикъл всяка дума от файла **sample.txt** и проверяваме дали тя съвпада с някоя дума от списъка **words**. За четенето на думите от файла отново използваме класа **Scanner**. При проверката игнорираме разликата между малки и големи букви**.** В случай на съвпадение с вече добавена дума увеличаваме броя на срещанията на съответната дума от списъка **words**. Броят на срещанията на думите съхраняваме в целочислен масив **wordsCount**, чиято размерност съвпада с броя на думите в списъка **words** (елементите на масива **wordsCount** съвпадат позиционно с елементите на списъка **words**).
3. Записваме резултата от така извършеното преброяване във файла **result.txt**, спазвайки формата, зададен в условието. За отваряне и писане във файла е удобно да използваме класа **PrintStream**.

Имплементация

Директно следваме стъпките, които идентифицирахме и ги реализираме. Получаваме следния сорс код:

WordsCounter.java

```
import java.util.*;
import java.io.*;

public class WordsCounter {
```

```
   public static void main(String[] args)
      throws FileNotFoundException {

      ArrayList<String> words = new ArrayList<String>();
      Scanner wordsFile = new Scanner(new File("words.txt"));
      while (wordsFile.hasNextLine()) {
         words.add(wordsFile.nextLine().toLowerCase());
      }
      wordsFile.close();

      int[] wordsCount = new int[words.size()];
      Scanner sampleFile = new Scanner(new File("sample.txt"));
      while (sampleFile.hasNext()) {
         String sampleWord = sampleFile.next().toLowerCase();
         for (String word : words) {
            if (sampleWord.contains(word)) {
               wordsCount[words.indexOf(word)]++;
            }
         }
      }
      sampleFile.close();

      PrintStream resultFile = new PrintStream("result.txt");
      for (String word : words) {
         resultFile.format("%s - %s%n", word,
            wordsCount[words.indexOf(word)]);
      }
      resultFile.close();
   }
}
```

Ефективност на решението

Май подценихме задачата и избързахме да напишем сорс кода. Ако се върнем към препоръките от главата "Как да решаваме задачи по програмиране", ще видим, че пропуснахме една важна стъпка: **избор на подходящи структури от данни**. Написахме кода като ползвахме първата възможна структура от данни, за която се сетихме, но не помислихме дали има по-добър вариант.

Време е да вмъкнем няколко думи за бързодействието (ефективността) на нашето решение. В повечето случаи така написаната програма ще работи достатъчно бързо за голям набор от входни данни, което я прави приемливо решение при явяване на изпит. Въпреки това е възможно да възникне ситуация, в която файлът **words.txt** съдържа много голям брой думи (примерно 10 000), което ще доведе до голям брой елементи на списъка **words**. Причината да се интересуваме от това е методът **indexOf(…)**, който използваме за намиране на индекса на дадена дума. Неговото

бързодействие е обратно пропорционално на броя на елементите на списъка и в този случай ще имаме осезаемо забавяне при работата на програмата. Например при 10 000 думи търсенето на една дума ще изисква 10 000 сравнения на двойки думи. Това ще се извърши толкова пъти, колкото са думите в текста, а те може да са много, да кажем 200 000. Тогава решението ще работи осезаемо бавно.

Можем да решим описания проблем като използваме хеш-таблица вместо целочисления масив **wordsCount** в горния код. Ще пазим в хеш таблицата като ключове всички думи, които срещаме в текста, а като стойности ще пазим колко пъти се среща съответната дума. По този начин няма да се налага последователно търсене в списъка **words**, защото хеш-таблицата имплементира значително по-бързо асоциативно търсене сред своите елементи. Можеше да се сетим за това, ако бяхме помислили за структурите от данни преди да се хвърлим да пишем сорс кода. Май трябваше да се доверим на методологията за решаване на задачи, а не да действаме както си знаем, нали?

Нека видим подобрения по този начин вариант на решението:

WordsCounter.java

```
import java.util.*;
import java.io.*;

public class WordsCounter {
   public static void main(String[] args)
      throws FileNotFoundException {

      ArrayList<String> words = new ArrayList<String>();
      Scanner wordsFile = new Scanner(new File("words.txt"));
      while (wordsFile.hasNextLine()) {
         words.add(wordsFile.nextLine().toLowerCase());
      }
      wordsFile.close();

      Hashtable<String, Integer> wordsCount =
         new Hashtable<String, Integer>();
      Scanner sampleFile = new Scanner(new File("sample.txt"));
      while (sampleFile.hasNext()) {
         String sampleWord = sampleFile.next().toLowerCase();
         for (String word : words) {
            if (sampleWord.contains(word)) {
               if (wordsCount.containsKey(word)) {
                  wordsCount.put(word, wordsCount.get(word) + 1);
               } else {
                  wordsCount.put(word, 1);
               }
```

```
            }
         }
      }
      sampleFile.close();

      PrintStream resultFile = new PrintStream("result.txt");
      for (String word : words) {
         int count = wordsCount.containsKey(word) ?
            wordsCount.get(word) : 0;
         resultFile.format("%s - %s%n", word, count);
      }
      resultFile.close();
   }
}
```

Тестване на решението

Разбира се, както при всяка друга задача, е много важно да тестваме решението, което сме написали и е препоръчително да измислим свои собствени примери освен този, който е даден в условието, и да се убедим, че изходът е коректен.

Трябва да тестваме и граничните случаи: какво става, ако единият от входните файлове е празен или и двата са празни? Какво става, ако в двата файла има само по една дума? Трябва да проверим дали малки и главни букви се считат за еднакви.

Накрая трябва да тестваме за скорост. За целта с малко copy/paste правим списък от 10 000 думи във файла **words.txt** и копираме текста от файла **sample.txt** достатъчно на брой пъти, за да достигне до 5-10 MB. Стартираме и се ужеждаваме, че имаме проблем. Чакаме минута-две, но програмата не завършва. Нещо не е наред.

Търсене на проблема с бързодействието

Ако пуснем програмата през дебъгера, ще се забележим, че имаме много глупава грешка в следния фрагмент код:

```
while (sampleFile.hasNext()) {
   String sampleWord = sampleFile.next().toLowerCase();
   for (String word : words) {
      if (sampleWord.contains(word)) {
         if (wordsCount.containsKey(word)) {
            wordsCount.put(word, wordsCount.get(word) + 1);
         } else {
            wordsCount.put(word, 1);
         }
      }
```

```
    }
}
```

Вижда се, че ако имаме 10 000 думи в масива **words** и 100 000 думи, които прочитаме една по една, за всяка от тях ще обходим във **for**-цикъл нашия масив и това прави 10 000 * 100 000 операции, които отнемат доста време. Как да оправим проблема?

Оправяне на проблема с бързодействието

За да работи коректно програмата очевидно трябва да преминем поне през веднъж през целия текст. Ако не прегледаме целия текст има опасност да не преброим някоя от думите. Следователно трябва да търсим ускорение на кода, който обработва всяка от думите. В текущата имплементация се върти цикъл до броя думи, които броим и ако те са много, този цикъл забавя чувствително програмата.

Идва ни идеята да заменим цикъла по думите, които броим с нещо по-бързо. Дали е възможно? Да помислим защо въртим този цикъл. Въртим го, за да видим дали думата, която сме прочели от текста е сред нашия списък от думи, за които броим колко пъти се срещат. Реално ни трябва бързо търсене в множество от думи. За целта може да се ползва или **HashSet** или **HashMap**, нали? Да си припомним структурите от данни множество и хеш-таблица. При тях може да се реализира изключително бързо търсене дори ако елементите са огромен брой.

Изводът е, че до момента сгрешихме на няколко пъти от прибързване. Ако бяхме помислили за структурите от данни и за ефективността преди да напишем кода, щяхме да си спестим много време и писане. Нека сега поправим грешката. Хрумва ни следната идея:

1. Правим си хеш-таблица и в нея записваме като ключове всички думи от файла **words.txt**. Като стойност в тези ключове записваме числото 0. Това е броят срещания на всяка дума в текста в началния момент, преди да сме започнали да го сканираме.
2. Сканираме текста дума по дума и търсим всяка от тях в хеш-таблицата. Това е бърза операция (търсене в хеш-таблица по ключ). Ако намерим думата, увеличаваме с 1 стойността в съответния ключ. Така си осигуряваме, че всяко срещане се отбелязва и накрая за всяка дума ще получим броя на срещанията ѝ.
3. Накрая сканираме думите от файла **words.txt** и за всяка търсим в хеш-таблицата колко пъти се среща в текста.

С новия алгоритъм при обработката на всяка дума от текста имаме по едно търсене в хеш-таблица и нямаме претърсване на масив, което е много бавна операция. Ето как изглежда новия алгоритъм:

FastWordsCounter.java

```
import java.util.*;
import java.io.*;

public class FastWordsCounter {
   public static void main(String[] args)
      throws FileNotFoundException {

      ArrayList<String> words = new ArrayList<String>();
      Hashtable<String, Integer> wordsCount =
         new Hashtable<String, Integer>();
      Scanner wordsFile = new Scanner(new File("words.txt"));
      while (wordsFile.hasNextLine()) {
         String word = wordsFile.nextLine().toLowerCase();
         words.add(word);
         wordsCount.put(word, 0);
      }
      wordsFile.close();

      Scanner sampleFile = new Scanner(new File("sample.txt"));
      while (sampleFile.hasNext()) {
         String word = sampleFile.next().toLowerCase();
         Integer count = wordsCount.get(word);
         if (count != null) {
            wordsCount.put(word, count + 1);
         }
      }
      sampleFile.close();

      PrintStream resultFile = new PrintStream("result.txt");
      for (String word : words) {
         int count = wordsCount.get(word);
         resultFile.format("%s - %s%n", word, count);
      }
      resultFile.close();
   }
}
```

Повторно тестване на проблема с бързодействието

Остава да тестваме новия алгоритъм: дали е коректен и дали работи бързо. Дали е коректен лесно можем да проверим с примерите, с които сме тествали и преди. Дали работи бързо можем да тестваме с големия пример (10 000 думи и 10 MB текст). Бързо се убеждаваме, че този път дори при големи обеми текстове програмата работи бързо. Дори пускаме 20 000 думи и 100 MB файл, за да видим дали ще работи. Уверяваме се, че дори и при

такъв обем данни програмата работи стабилно и с приемлива скорост (20-30 секунди на компютър от 2008 г.).

Задача 3: Училище

В едно училище учат ученици, които са разделени в учебни групи. На всяка група преподава един учител.

За учениците се пази следната информация: име и фамилия.

За всяка група се пази следната информация: наименование и списък на учениците.

За всеки учител се пази следната информация: име, фамилия и списък от групите, на които преподава. Един учител може да преподава на повече от една група.

За училището се пази следната информация: наименование, списък на учителите, списък на групите, списък на учениците.

1. Да се проектира съвкупност от класове с връзки между тях, които моделират училището.
2. Да се реализират методи за добавяне на учител, за добавяне на група и за добавяне на ученик. Списъците могат да се представят чрез масиви или чрез списъчни структури.
3. Да се реализира метод за отпечатване на информация за даден учител: име, фамилия, списък на групите, на които преподава, и списък на учениците от всяка от тези групи.
4. Да се напише примерна тестова програма, която демонстрира работата на реализираните класове и методи.

Пример:

Училище "Свобода". Учители: Димитър Георгиев, Христина Николова.

Група "английски език": Иван Петров, Васил Тодоров, Елена Михайлова, Радослав Георгиев, Милена Стефанова, учител Христина Николова.

Група "френски език": Петър Петров, Васил Василев, учител Христина Николова.

Група "информатика": Милка Колева, Пенчо Тошев, Ива Борисова, Милена Иванова, Христо Тодоров, учител Димитър Георгиев.

Решение на задачата

Това е добър пример за задача, чиято цел е да тества умението на кандидатите, явяващи се на изпита да използват ООП за моделиране на задачи от реалния свят. Ще моделираме предметната област като дефинираме взаимно свързаните класове **`Student`**, **`Group`**, **`Teacher`** и **`School`**. За да бъде изцяло изпълнено условието на задачата ще имаме нужда и от клас

SchoolTest, който демонстрира работата на дефинираните от нас класове и методи.

Измисляне на идея за решение

В тази задача няма нищо за измисляне. Тя не е алгоритмична и в нея няма какво толкова да мислим. Трябва за всеки обект от описаните в условието на задачата (студенти, учители, ученици, училище и т.н.) да дефинираме по един клас и след това в този клас да дефинираме свойства, които го описват и действия, които той може да направи. Това е всичко.

Разделяме задачата на подзадачи

Имплементацията на всеки един от класовете можем да разглеждаме като подзадача на дадената:

- Клас за студентите – **Student**
- Клас за групите – **Group**
- Клас за учителите – **Teacher**
- Клас за училището – **School**
- Клас за тестване на останалите класове с примерни данни – **SchoolTest**

Имплементиране: стъпка по стъпка

Удачно е да започнем реализацията с класа **Student**, тъй като от условието на задачата лесно се вижда, че той не зависи от останалите три.

Класът Student

В дефиницията имаме само две полета, представляващи име и фамилия на ученика и метод **getName()**, който връща низ с името на ученика. Дефинираме го по следния начин:

Student.java

```
public class Student {
   private String firstName;
   private String lastName;

   public Student(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public String getName() {
```

```
      return this.firstName + " " + this.lastName;
   }
}
```

Класът Group

Следващият клас, който дефинираме е **Group**. Избираме него, защото в дефиницията му се налага да използваме единствено класа **Student**. Полетата, които ще дефинираме представляват име на групата и списък с ученици, които посещават групата. За реализацията на списъка с ученици ще използваме класа **ArrayList**. Класът ще има методи **getName()** и **getStudents()**, които извличат стойностите на двете полета. Добавяме още два метода, които ни трябват – **addStudent(…)** и **printStudents(…).** Методът **addStudent(…)** добавя обект от тип **Student** към списъка **students**, а методът **printStudents(…)** отпечатва името на групата и имената на учениците в нея. Нека сега видим цялата реализация на класа:

Group.java

```
import java.util.ArrayList;
import java.io.PrintStream;

public class Group {
   private String name;
   private ArrayList<Student> students;

   public Group(String name) {
      this.name = name;
      this.students = new ArrayList<Student>();
   }

   public String getName() {
      return this.name;
   }

   public ArrayList<Student> getStudents() {
      return this.students;
   }

   public void addStudent(Student student) {
      students.add(student);
   }

   public void printStudents(PrintStream output) {
      output.printf("Group name: %s%n", this.name);
      output.printf("Students in group:%n");
      for(Student student : this.students) {
```

```
            output.printf("  Name: %s%n", student.getName());
        }
    }
}
```

Класът Teacher

Нека сега дефинираме класа **Teacher**, който използва класа **Group**. Неговите полета са име, фамилия и списък с групи. Той има методи **addGroup(…)** и **printGroups(…)**, аналогични на тези в класа **Group**. Методът **printGroups(…)** отпечатва името на учителя и извиква метода **printStudents(…)** на всяка група от списъка с групи:

Teacher.java

```
import java.util.ArrayList;
import java.io.PrintStream;

public class Teacher {
   private String firstName;
   private String lastName;
   private ArrayList<Group> groups;

   public Teacher(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.groups = new ArrayList<Group>();
   }

   public void addGroup(Group group) {
      this.groups.add(group);
   }

   public void printGroups(PrintStream output) {
      output.printf("Teacher name: %s %s%n", this.firstName,
         this.lastName);
      output.printf("Groups of teacher:%n");
      for(Group group : this.groups) {
         group.printStudents(output);
      }
   }
}
```

Класът School

Завършваме обектния модел с дефиницията на класа **School**, който използва всички вече дефинирани класове. Полетата му са име, списък с учители, списък с групи и списък с ученици. Методите **getName()** и

getTeachers() използваме за извличане на нужните данни. Дефинираме методи **addTeacher(…)** и **addGroup(…)** за добавяне на съответните обекти**.** За удобство при създаването на обектите, в метода **addGroup(…)** имплементираме следната функционалност: освен добавянето на самата група като обект, добавяме към списъка с ученици и учениците, които попадат в тази група (но все още не са добавени в списъка на училището). Ето и целия код на класа:

School.java

```
import java.util.ArrayList;

public class School {
   private String name;
   private ArrayList<Teacher> teachers;
   private ArrayList<Group> groups;
   private ArrayList<Student> students;

   public School(String name) {
      this.name = name;
      this.teachers = new ArrayList<Teacher>();
      this.groups = new ArrayList<Group>();
      this.students = new ArrayList<Student>();
   }

   public String getName() {
      return name;
   }

   public ArrayList<Teacher> getTeachers() {
      return this.teachers;
   }

   public void addTeacher(Teacher teacher) {
      teachers.add(teacher);
   }

   public void addGroup(Group group) {
      groups.add(group);
      for(Student student : group.getStudents()) {
         if(!this.students.contains(student)) {
            this.students.add(student);
         }
      }
   }
}
```

Класът TestSchool

Следва реализацията на класа **SchoolTest**, който има за цел да демонстрира класовете и методите, които дефинирахме. Това е и нашата последна подзадача – с нея решението е завършено. За демонстрацията използваме данните от примера в условието:

SchoolTest.java

```
public class SchoolTest {
   public static void addObjectsToSchool(School school) {
      Teacher teacherGeorgiev = new Teacher("Димитър", "Георгиев");
      Teacher teacherNikolova = new Teacher("Христина", "Николова");

      school.addTeacher(teacherGeorgiev);
      school.addTeacher(teacherNikolova);

      // Add the English group
      Group groupEnglish = new Group("английски език");
      groupEnglish.addStudent(new Student("Иван", "Петров"));
      groupEnglish.addStudent(new Student("Васил", "Тодоров"));
      groupEnglish.addStudent(new Student("Елена", "Михайлова"));
      groupEnglish.addStudent(new Student("Радослав", "Георгиев"));
      groupEnglish.addStudent(new Student("Милена", "Стефанова"));
      groupEnglish.addStudent(new Student("Иван", "Петров"));

      school.addGroup(groupEnglish);
      teacherNikolova.addGroup(groupEnglish);

      // Add the French group
      Group groupFrench = new Group("френски език");
      groupFrench.addStudent(new Student("Петър", "Петров"));
      groupFrench.addStudent(new Student("Васил", "Василев"));

      school.addGroup(groupFrench);
      teacherNikolova.addGroup(groupFrench);

      // Add the Informatics group
      Group groupInformatics = new Group("информатика");
      groupInformatics.addStudent(new Student("Милка",
         "Колева"));
      groupInformatics.addStudent(new Student("Пенчо", "Тошев"));
      groupInformatics.addStudent(new Student("Ива", "Борисова"));
      groupInformatics.addStudent(new Student("Милена", "Иванова"));
      groupInformatics.addStudent(new Student("Христо", "Тодоров"));

      school.addGroup(groupInformatics);
      teacherGeorgiev.addGroup(groupInformatics);
```

```
   }

   public static void main(String[] args) {
      School school = new School("Свобода");

      addObjectsToSchool(school);

      for(Teacher teacher : school.getTeachers()) {
         teacher.printGroups(System.out);
         System.out.println();
      }
   }
}
```

Изпълняваме програмата и получаваме очаквания резултат:

```
Teacher name: Димитър Георгиев
Groups of teacher:
Group name: информатика
Students in group:
  Name: Милка Колева
  Name: Пенчо Тошев
  Name: Ива Борисова
  Name: Милена Иванова
  Name: Христо Тодоров

Teacher name: Христина Николова
Groups of teacher:
Group name: английски език
Students in group:
  Name: Иван Петров
  Name: Васил Тодоров
  Name: Елена Михайлова
  Name: Радослав Георгиев
  Name: Милена Стефанова
  Name: Иван Петров
Group name: френски език
Students in group:
  Name: Петър Петров
  Name: Васил Василев
```

Разбира се, в реалния живот програмите не тръгват от пръв път, но в тази задача грешките, които можете да допуснете, са тривиални и няма смисъл да ги дискутираме. Всичко е въпрос на написване (ако познавате работата с класове и обектно-ориентираното програмиране като цяло).

Тестване на решението

Остава, както при всяка задача, да тестваме дали решението работи правилно. Ние вече го направихме. Може да направим и няколко теста с гранични данни, примерно група без студенти, празно училище и т.н. тестове за бързодействие няма да правим, защото задачата има неизчислителен характер. Това е всичко.

Упражнения

1. Напишете програма, която отпечатва спирална квадратна матрица, започвайки от числото 1 в горния десен ъгъл и движейки се по часовниковата стрелка. Примери при N=3 и N=4:

7	8	1
6	9	2
5	4	3

10	11	12	1
9	16	13	2
8	15	14	3
7	6	5	4

2. Напишете програма, която брои думите в текстов файл, но за дума счита всяка последователност от символи (подниз), а не само отделените с разделители. Например в текста "Аз съм студент в СофтУни" подниза "с", "сту", "а" и "аз съм" се срещат съответно 3, 1, 2 и 1 пъти.
3. Моделирайте със средствата на ООП файловата система в един компютър. В нея имаме устройства, директории и файлове. Устройствата са примерно твърд диск, флопи диск, CD-ROM устройство и др. Те имат име и дърво на директориите и файловете. Една директория има име, дата на последна промяна и списък от файлове и директории, които се съдържат в нея. Един файл има име, дата на създаване, дата на последна промяна и съдържание. Файлът се намира в някоя от директориите. Файлът може да е текстов или бинарен. Текстовите файлове имат за съдържание текст (**`String`**), а бинарните – поредица от байтове (**`byte[]`**). Направете клас, който тества другите класове и показва, че с тях можем да построим модел на устройствата, директориите и файловете в компютъра.
4. Ползвайки класовете от предната задача с търсене в Интернет напишете програма, която взима истинските файлове от компютъра и ги записва във вашите класове (без съдържанието на файловете, защото няма да ви стигне паметта).

Решения и упътвания

1. Задачата е аналогична на първата задача от примерния изпит по програмиране. Можете да модифицирате примерното решение, дадено по-горе.

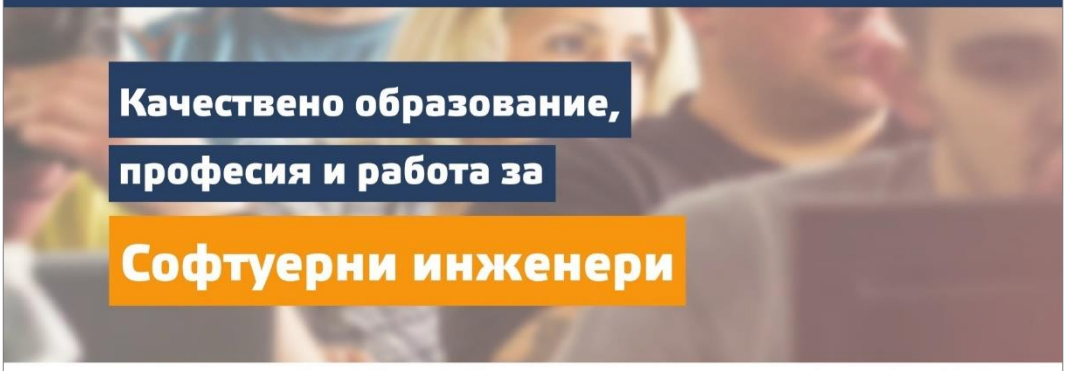
2. Трябва да сканирате текста не дума по дума, ами буква по буква и след всяка следваща буква да я долепяте към текущия буфер **`buf`** и да проверявате всяка от търсените думи за съвпадение с **`endsWith()`**. Разбира се, няма да можете да ползвате ефективно хеш-таблица и ще имате цикъл по думите за всяка буква от текста, което не е най-бързото решение.

   Реализирането на бързо решение изисква използването на сложна структура от данни, наречена **суфиксно дърво**. Можете да потърсите в Google следното: **`"suffix tree" "pattern matching" filetype:ppt`**.

3. Задачата е аналогична на задачата с училището от примерния изпит по програмиране и се решава чрез същия подход. Дефинирайте класове **`Device`**, **`Directory`**, **`File`**, **`ComputerStorage`** и **`ComputerStorageTest`**. Помислете какви свойства има всеки от тези класове и какви са отношенията между класовете. Когато тествате слагайте примерно съдържание за файловете (примерно по 1 думичка), а не оригиналното, защото то е много обемно. Помислете може ли един файл да е в няколко директории едновременно.

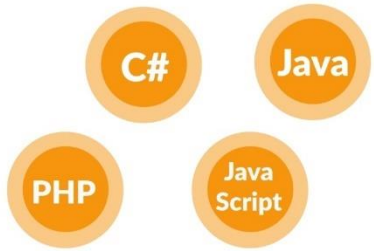
4. Използвайте класа **`java.io.File`** и методите **`listRoots()`**, **`listFiles()`** и **`isDirectory()`**.

SOFTWARE UNIVERSITY



Качествено образование, професия и работа за Софтуерни инженери

- Безплатен старт за начинаещи - **присъствено и онлайн**
- Избор измежду **най-търсените професии** в софтуерната индустрия
- Съдействие за **кариерен старт**
- Отлични преподаватели, ментори и студентска общност



"Софтуерният университет" (СофтУни) е основан с идеята за иновативен и модерен образователен център, който създава истински **професионалисти в света на програмирането**.

СофтУни предлага цялостна програма по софтуерно инженерство с **най-търсените софтуерни технологии** и най-модерните учебни практики. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофтУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ

Кандидатствай softuni.bg/**apply**

Заключение

Ако сте стигнали до заключението и сте прочели внимателно цялата книга, приемете нашите **заслужени поздравления**! Убедени сме, че сте научили ценни знания за принципите на програмирането, които **ще ви останат за цял живот**. Дори да минат години, дори технологиите да се променят и компютрите да не бъдат това, което са в момента, фундаменталните знания за структурите от данни в програмирането и алгоритмичното мислене винаги ще ви служат неизменно, ако се занимавате с програмиране или ИТ.

Ако освен, че сте прочели внимателно цялата книга, сте **решили и всички задачи от упражненията** към всяка от главите, вие можете гордо да се наречете програмист. Всяка технология, с която ще се захванете от сега нататък, ще ви се стори **лесна като детска игра**. След като сте усвоили основите и фундаменталните принципи на програмирането, със завидна лекота ще се научите да ползвате бази данни и SQL, да разработвате уеб приложения и сървърен софтуер, да програмиране за мобилни устройства и каквото още поискате. Вие **имате огромно предимство пред мнозинството от практикуващите програмиране**, които не знаят какво е хеш-таблица, как работи търсенето в дървовидна структура и какво е сложност на алгоритъм. Ако наистина сте се блъскали да решите всички задачи от цялата книга, със сигурност сте постигнали едно завидно ниво на фундаментално разбиране на концепциите на програмирането и правилното мислене на програмист, което ще ви помага години наред.

Ако не сте решили всичките задачи от **упражненията** или поне голямата част от тях, **върнете се и ги решете**. Да, отнема много време, но това е начинът да се научите да програмирате – чрез много труд и усилия. Без да практикувате много сериозно програмирането, няма да го научите!

Може би се чудите с какво да продължите развитието си като софтуерен инженер, на което с тази книга сте поставили здрави основи. Можем да ви дадем следните насоки, към които да се ориентирате:

1. Изберете **език + платформа за програмиране**, примерно Java + Java EE, C# + ASP.NET MVC, JavaScript + Node.js, PHP + Symfony, Python + Django или Ruby + Rails. Няма проблем, ако решите да не продължите с езика Java. Фокусирайте се върху технологиите, които платформата ви предоставя.
2. Прочетете някоя книга за **бази данни** и се научете да моделирате данните на вашето приложение с таблици и връзки между тях. Научете се как да построявате заявки за извличане и промяна на данните чрез езика SQL. Научете се да работите с някой сървър за

релационни бази данни, примерно Oracle, SQL Server или MySQL. Разгледайте и нерелационните бази данни, например MongoDB.

3. Научете някоя технология за изграждане на **динамични уеб приложения**. Започнете с някоя книга или видео уроци за **HTML**, **CSS** и **JavaScript**. След това разгледайте какви средства за създаване на уеб приложения предоставя вашата любима платформа, примерно Servlets / JSP / JSF при Java платформата или ASP.NET при .NET платформата или пък захванете Python и Django. Научете се да правите прости уеб сайтове с динамично съдържание.

4. Напишете **мобилно приложение** за таблет или телефон. Най-лесно ще ви е ако се захванете с платформата **Android**, защото тя е базирана на Java. Опитайте да направите например игра за телефон или друго приложение, което използва **графичен потребителски интерфейс** (GUI). Така ще се научите да работите с библиотеки за потребителски интерфейс.

5. Захванете се да напишете **някакъв по-сериозен проект**, примерно Интернет магазин, софтуер за обслужване на склад или магазин. Това ще ви даде възможност да се сблъскате с реалните проблеми от реалната разработка на софтуер. Ще добиете много ценен реален опит и ще се убедите, че писането на сериозен софтуер е много по-трудно от писането на прости програмки.

6. **Започнете работа в софтуерна фирма**. Ако наистина сте решили всички задачи от тази книга, лесно ще ви предложат работа. Работейки по реални проекти ще научите страхотно много нови технологии от колегите си и ще се убедите, че макар и да знаете много за програмирането, сте едва в началото на развитието си като софтуерен инженер. Само при **работа по проекти** съвместно с колеги ще се сблъскате с проблемите при работа в екип и с практиките и инструментите за ефективно преодоляване на тези проблеми. Ще трябва да поработите поне няколко години, докато се утвърдите като специалист по разработка на софтуер.

Можете **да си спестите много труд, време и нерви**, ако решите да преминете през всички описани стъпки от развитието си като софтуерен инженер в **Софтуерния университет (СофтУни)** под ръководството на инструктори с реален опит в софтуерната индустрия. **СофтУни** е най-лесният начин да поставите основите на изграждането си като софтуерен инженер, но не е единственият начин. Всичко зависи от вас!

От името на целия авторски колектив ви пожелаваме **неспирни успехи в професията и в живота**!

Светлин Наков,
София, 14.12.2008 г.
(последна редакция: 7.04.2017 г.)

Това без съмнение е една от най-добрите книги по програмиране за начинаещи и покрива фундаментални знания, които ще ползвате през цялата си кариера на софтуерни разработчици. Със сигурност бих искал да имам книга като тази, когато самият аз навлизах в програмирането.

Васил Поповски, софтуерен архитект, VMware

Една прекрасна книга, която изчерпателно запознава читателя, с основите на програмирането, използвайки най-популярния език Java. Задължително четиво за всеки търсещ успешна професионална реализация като програмист. Това е вашата книга за програмиране!

Драгомир Николов, мениджър разработка, Software AG

Ако искате с първата си книга по програмиране да добиете солидни знания, които ще са ви полезни години наред, то това трябва да е вашата първа книга!

Веселин Райчев, софтуерен инженер, Google

Книгата "Въведение в програмирането с Java" съдържа фундаменталните първоначални познания, от които всеки начинаещ програмист има нужда, представени в компактен вид, даващ възможност за лесно и бързо усвояване.

Явор Ташев, софтуерен инженер, Microsoft

Началото винаги е трудно, а може да бъде и още по-трудно – достатъчно е да не знаеш от къде да почнеш или да си създадеш грешен подход. Тази книга е за всички, които сега започват и искат да бъдат добри програмисти, а също и за всички самоусъвършенстващи се програмисти, желаещи да запълнят пропуските, които със сигурност имат.

Любомир Иванов, ръководител на отдел, Mobiltel

Настоящата книга е едно изключително добро въведение в програмирането за начинаещи и водещ пример в течението (промоцирано от Wikipedia и други) да се създава и разпространява достъпно за всеки знание не само безплатно, но и с изключително високо качество.

Панайот Добриков, програмен директор, SAP AG и автор на книгата "Програмиране=++Алгоритми"

АВТОРИТЕ:

Борис Вълков
Веселин Колев
Владимир Цанев
Данаил Алексиев
Лъчезар Божков
Лъчезар Цеков
Марин Георгиев
Марио Пешев
Мариян Ненчев
Михаил Стойнов
Николай Василев
Николай Недялков
Петър Велев
Радослав Иванов
Румяна Топалска
Стефан Стаев
Светлин Наков
Теодор Стоев
Христо Тодоров
Цвятко Конов

Уеб сайт:
http://www.introprogramming.info

SOFTWARE UNIVERSITY

ISBN 978-954-400-055-4