

Ангелин Лалев

ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО НА **JAVA**



АНГЕЛИН ЛАЛЕВ

**ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО
НА JAVA**

2023

Въведение в програмирането на JAVA

Онлайн издание.

Copyright© Ангелин Лазаров Лалев; Автор: Ангелин Лазаров Лалев

ISBN: 978-619-91424-2-7

Настоящото електронно издание е лицензирано под Creative Commons CC-BY-NC-ND 4.0 лиценз. Пълният текст на лиценза се намира на следния адрес:

<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

СЪДЪРЖАНИЕ

ПРЕДГОВОР	3
Глава 1. Запознаване с Java.....	3
1. История и версии на Java.....	3
2. Компоненти на Java	3
3. „Hello World“ приложение	6
4. Работа с java, javac и jar от командния ред	10
4.1. Работа с javac	11
4.2 Работа с java	19
4.3. Работа с jar	20
5. Създаване на Hello World приложение с помощта на интегрирана среда за разработка.....	22
Глава 2. Елементарни типове, променливи и литерали.....	36
1. Елементарни типове в Java.....	36
2. Променливи.....	41
3. Литерали.....	43
4. Понятие за референтни типове	49
5. Автоматични превръщания между елементарни типове	51
Глава 3. Оператори в Java	59
1. Аритметични оператори	59
2. Оператор за сцепване на низове.....	60
3. Оператор за присвояване	61
4. Оператори за сравнение.....	63
5. Логически оператори	65

4. Побитови оператори.....	70
5. Оператори за побитово изместване	72
6. Троичен оператор	75
7. Оператори за увеличаване / намаляване с единица	78
8. Комбинирани оператори за присвояване	79
9. Приоритет на операторите в Java.....	80
10. Превръщания между стойности в контекста на изчисляване на изрази....	83
Глава 4. Условни конструкции и цикли	86
1. Конструкция if-else.....	86
2. Конструкция switch	98
3. Цикли с брояч	106
4. Цикли с while и do-while.....	115
Глава 5. Работа със символни низове	118
1. Символни променливи и символни литерали	118
2. Методи за работа със символни низове	121
2.1. Номериране на позициите в символните низове	123
2.2. Методи length и isEmpty	124
2.3. Методи charAt, indexOf и lastIndexOf.....	125
2.4. Методи toLowerCase и toUpperCase	126
2.5. Методи startsWith и endsWith.....	127
2.6. Методи trim, strip, stripLeading и stripTrailing.....	128
2.7. Метод substring	129
2.8. Метод split	129
2.9. Метод replace	132

2.10. Метод contains.....	133
3. Представяне на символните низове в Java.....	134
Глава 6. Масиви	140
1. Понятие за масив	140
2. Деклариране и инициализиране на масиви	140
3. Многомерни масиви	144
6. Свойства на масивите	149
5. Класът java.lang.Arrays	156
Глава 7. Методи.....	163
1. Въведение в методите	163
2. Използване на методи в Java	174
Глава 8. Въведение в обектно-ориентираното програмиране.....	186
1. Инкапсулация.....	187
1.1. Инкапсулация на полета	189
1.2. Инкапсулация на методи.....	192
1.3. Роля на конструкторите за постигане на инкапсулация	196
1.4. Пакети.....	201
2. Наследяване	204
3. Полиморфизъм.....	215
4. Скриване.....	221
4.1. Скриване на ниво класове	222
4.2. Скриване на ниво пакети	229
Глава 9. Разширено обектно-ориентирано програмиране с Java.....	231
1. Разширена инкапсулация	231

1.1. Статични методи и полета	231
1.2. Инициализационни блокове и директна инициализация	236
1.3. Изброими типове	238
1.4. Анонимни класове, локални класове, вътрешни класове и статични вложени класове	242
3.1. Статични вложени класове	243
3.2. Вътрешни класове	246
2. Разширено скриване.....	247
2. Модификатори за достъп в Java	261
5. Използване на super и this	276
6. Конструктори	280

ПРЕДГОВОР

Едно от най-трудните неща за начинаещите програмисти е да си представят мястото на изучаваната технология или програмен език в „голямата картинка“ на нещата. Не случаен фактът, че при всичката сложност и цялото многообразие на ИТ технологиите, огромна част от начинаещите са силно привлечени от HTML и CSS. При тях инвестираните усилия са минимални, докато видимият резултат е най-голям.

Независимо дали го осъзнават или не, всички програмисти правят своеобразен анализ „ползи-разходи“ на всяка стъпка от изучаването на нов програмен език. За начинаещите, които много трудно могат да осмислят цялата сложност, която стои зад невидимите „бак-енд“ услуги, Java със сигурност изглежда излишно труден за изучаване език за програмиране. Това е и причината много от тях да се насочат към по-лесни езици за програмиране като JavaScript, Python, PHP и Kotlin.

За съжаление, въпреки че това е вероятно най-важната задача пред въведението към подобна книга, мъдростта, ефективността и философията на Java не могат да бъдат разяснени в рамките на няколко параграфа. Косвените доказателства за предимствата на Java, като например това, че към 2023 тя се използва в 90% от Fortune 500 компаниите, или фактът, че за последните 25 години Java винаги е поставяна на първите места по популярност сред програмните езици, също трудно биха убедили начинаещите, че си струва да отделят поне два пъти повече време и усилия, за да научат Java вместо например Python.

Предимствата на Java трябва да бъдат „изпитани“ от практикуващия в редица учебни и реални сценарии с постепенно нарастваща сложност. На този етап на въведението авторът може само да даде наистина мощна аналогия за предимствата и подхода на Java.

Всички ние като деца сме строили замъци от пясък или Lego блокове и сме напълно наясно колко забавно и удовлетворяващо е усещането, че създаваме нещо уникално със собствените си ръце. Когато преди много години авторът, в тийнейджърските си години, се научи да програмира на Pascal, усещането в много отношения беше същото. Както много други той имаше и удоволствието, и неудоволствието постепенно да осъзнае, с увеличаване на сложността на задачите и проектите, че разработката на професионален софтуер в повечето случаи не е детска игра.

Също както пясъкът и Lego блоковете естествено ограничават височината на това, което може да се построи с тях, някои езици за програмиране са по-добри от други, когато изведнъж се изправим пред проблеми, възникващи от наличието на стотици хиляди редове програмен код, разработвани от различни хора и обременени от безброй изисквания за ефективност, срокове, бързодействие и сигурност.

За разлика от езици като PHP и JavaScript, които могат да изтъкнат за свое най-голямо предимство това, че поне първоначално те са били създадени, за да улеснят писането на малки проекти, разработвани от непрофесионални програмисти, Java е създадена, за да подпомогне именно професионалистите. От създаването си до днес Java прави това много успешно, там, където кодовата база е твърде голяма, там, където приложенията са критични за работата на организацията, както и там, където се изисква перфектен баланс между бързодействие, сигурност и ценова ефективност.

Връщайки се към аналогията със строителството, не можем да не се съгласим, че преминавайки от пясъчни замъци към небостъргачи, ние все пак трябва да се откажем от нещо. Създаването на небостъргачи е по-скучен процес от създаването на пясъчни замъци. С повишаването на височината, броят на възможните успешни решения намалява, а възможността за грешки расте експоненциално, което елиминира до някаква степен елемента на експериментиране и забавление от креативния процес.

Java и центрираната върху нея екосистема определено е повлияна по подобен начин от факта, че тя е насочена към по-големи и сериозни проекти. Синтаксисът на Java много пъти е прекалено експлицитен и пълен с идиосинкратизми¹. Рамките за програмиране на приложения на Java са големи и „претоварени“ с безброй много функции. Програмният код трябва да се компилира, преди да се използва. При добавянето на нови функции към езика, създателите на Java трябва да се погрижат за осигуряване на максимална съвместимост с вече написаните милиони и милиони редове програмен код. И не на последно място - за да се научим да работим с Java и нейната екосистема, се налага да усвоим много информация.

За разлика от много други учебници, които се опитват да опазят начинаещите от ранни разочарования и преподават елементи на програмиране, без да влизат в детайлите и спецификите на Java, настоящата книга се опитва да представи езика Java по максимално честен начин. Макар че авторът е вложил усилия изложението да бъде максимално последователно, дискусиата на трудните теми, като например правилата за типово преобразуване и работата с типови шаблони, не се избягва. Тези теми се представят в първия възможен момент. Макар това да прави „кривата на обучение“ малко по-стръмна, авторът със сигурност няма намерение да се извинява за този подход, като е на мнение, че това помага много за ранното излагане на философията на езика пред обучаемите.

Практиката на автора като преподавател показва, че учебникът със сигурност може да се използва ефективно от начинаещи, които за пръв път се занимават с програмиране, като също така е много полезна и на обучаеми, които са преминали и усвоили основите на програмирането на Java или друг програмен език. Читателите, които за пръв път се занимават с програмиране, трябва да знаят,

¹ Несъгласните с това твърдение могат да например да проверят спецификацията на Java за ефектите от премахването на типовите шаблони при компилация върху извикването на методи с подобни сигнатури или например правилата за аутобоксинг и ънбоксинг.

че няма да могат да прочетат материала в учебника „линейно“ от началото до края. Много често те ще трябва да пропуснат или преминат през определени по-сложни теми и да се върнат към тях, когато са готови. Ако не друго, след прочитането на книгата тези читатели ще знаят, че дискутираните трудни проблеми съществуват, и ще могат правилно да насочат последващото си обучение в тази посока.

Както при всички други учебници за програмиране, е изключително важно читателите да възпроизведат колкото се може повече от примерите към материала сами. Процесът на възпроизвеждане и внасяне на собствени модификации в примерите ще разкрие къде читателят е разбрал дадена концепция и къде той трябва да търси допълнителна информация.

Настоящият учебник е организиран в 9 глави:

Глава 1 е посветена на запознаване с инструментите на езика Java. В главата са представени детайли относно виртуалната машина и основните инструменти на Java, като е обърнато внимание на структурата на изходната и на компилираната програма. По-скоро теоретично, това изложение е безценно въведение в механизмите „под капака“ на Java, които неминуемо напомнят за себе си при разработването на всеки професионален проект. От главата отсъстват типичните инструкции за инсталирането на Java и типичните интегрирани среди за програмиране. Тази информация се изменя доста бързо и авторът е на мнение, ако тази информация бе включена в изложението, тя щеше да остане в рамките на месеци. Начинаещите програмисти могат да намерят актуална информация на сайта на EclipseFoundation и проекта Temurin, както и в безброй безплатни видеоматериали, публикувани в Youtube.

Глава 2 съдържа задълбочена дискусия на примитивните типове в Java. В главата са разгледани основните 7 примитивни типа и начините за деклариране на локални променливи. Отделено е значително внимание на ограниченията на всеки тип, както и на правилата за преобразуване между елементарни типове в Java.

Глава 3 запознава читателите с операторите в Java и техният приоритет. Отделено е внимание на аритметичните, логическите и побитовите оператори, както и операторите за увеличаване и намаляване с единица и операторите за битово изместване. В тази глава са разгледани и правилата за преобразуване на типове при изчисляване на изрази, тъй като те понякога са специфични за използвания оператор.

Глава 4 разглежда условните конструкции и конструкциите за осъществяване на цикли, които стоят в основата на всички програми. Разгледани са циклите с **for**, **while** и **do-while**, както и еквивалента на **for-each** в Java.

Глава 5 е посветена на работата със символни низове. В главата са разгледани не само функциите на класа **String**, но и начините, по които компилаторът на Java избягва повторението на символни низове в програмите и последствията за програмистите от тях.

Глава 6 излага работата с масиви в Java. В главата са изложени детайли за работа с масиви, като е акцентувано на особеностите, които произтичат от това, че масивите са референтни типове.

Глава 7 е посветена на представянето на методите в Java. Като ролята на тази глава е да играе ролята на въведение в материала, посветен на ООП.

Глава 8 представя основите на обектно-ориентираното програмиране, като се опитва да представи ООП като сбор от основните му принципи, а именно инкапсулация, наследяване, полиморфизъм и скриване. В тази глава целенасочено се избягва дискусиите на спецификите на Java, освен там, където това е неизбежно.

Глава 9 има за цел да разшири знанията на читателите за ООП като разшири и обогати разбирането на базовите принципи със специфики, които често са уникални за Java.

Авторът би се радвал да получи всякаква обратна връзка, включително и коментари за грешки и неточности в книгата на адреси **lalev@uni-svishtov.bg** и **lalev.angelin@gmail.com**.

Свищов,

юни 2023 г.

Глава 1. Запознаване с Java

1. История и версии на Java

Програмният език Java е създаден през периода от 1991 до 1995 година от Sun Microsystems. Версия 1.0 на езика излиза официално през 1996 година.

Java е един от първите езици, които в пълна степен прилагат идеите на обектно-ориентираното програмиране, както и първият програмен език в историята, който използва компилиране към платформено-независим байт-код. Поради предимствата, произтичащи от тези иновации, Java бързо се утвърждава като водещ език за програмиране със силни позиции при бизнес-ориентирания софтуер.

Въпреки че в миналото програмният език Java и неговите реализации бяха обременени от корпоративни лицензи, днес Java се разработва като софтуер с отворен код. Като много други проекти с отворен код, Java се разработва инкрементално, като на всеки 6 месеца се издава нова версия на компилатора и инструментите. Всяка 6-та версия се обявява за версия с дългосрочна поддръжка (“Long Time Support” или “LTS”).

Най-актуалната версия на Java към момента на писане на настоящето е Java 19 (от септември 2022). Последната LTS версия на Java е 17 (от септември 2021), като следващата LTS версия се очаква към септември 2024.

2. Компоненти на Java

Програмният език Java се състои от два основни компонента и множество допълнителни модули, които улесняват разработката и изпълнението на програми на Java.

Първият основен компонент на Java е компилаторът `javac`. `javac` проверява изходния код на програмите за валидност и генерира изпълнимите програми на Java. За разлика от класическите компилирани езици като C и C++,

които създават машинен код, оформен в специфичен за всяка отделна операционна система формат на изпълним файл (например EXE за Windows или ELF за Linux), компилаторът на Java създава платформено независим байт-код. Това позволява на програмите на Java да се изпълняват без прекомпиляция на широк кръг от компютърни системи, които използват коренно различни по организация и възможности процесори, и които работят под управлението на най-различни операционни системи. Изпълнимият файл с компилатора на Java е **javac.exe** под Windows и **javac** под останалите операционни системи.

За да се изпълни байткодът, създаден от компилатора на Java, обаче е необходим допълнителен компонент, който да адаптира байткода към особеностите на всяка конкретна комбинация от хардуер и операционна система. Тази роля се поема от **втория основен компонент на Java – т.нар. „виртуалната машина“**.

Виртуалната машина служи за изпълнение на програмите под Java и е платформено специфична. Тоест, за да може програми на Java да се изпълняват на даден нов процесор или под управлението на нова операционна система, първо трябва да бъде разработена специфична адаптация на виртуалната машина за новия процесор или операционна система. За щастие Java е добре поддържана и виртуални машини съществуват за почти всички възможни комбинации от общоцелеви процесори и операционни системи.

Изпълнимият файл на виртуалната машина е **java.exe** под Windows и **java** при всички останали операционни системи.

В миналото Java се предлагаше в инсталационни пакети от два основни типа. Така нареченият Java Development Kit (JDK) съдържаше всички компоненти на Java и бе подходящ както за създаване и компилиране на програми на Java, така и за изпълнение на тези програми. Тъй като JDK заемаше относително много дисково пространство, и тъй като повечето потребители на Java се интересуваха от изпълнението на готови програми, а не от тяхната разработка, съществуваше специален инсталационен пакет, който инсталираше

само виртуалната машина и стандартната библиотека от класове на езика. Тази комбинация не съдържаеше компилатора и бе достатъчна за стартирането на програми на Java, но не и за разработката им. Този пакет се наричаше Java Runtime Environment (JRE).

В последните години повечето реализации на Java спряха да предлагат отделно JRE. Вместо това Java получи специален инструмент – **jlink** – който позволи на разработчика на програмата да избира кои модули от JDK и стандартната библиотека с класове да бъдат комплектовани заедно с виртуалната машина на Java според нуждата на програмата. Този по-гъвкав подход позволи създаването на „параметризирани“ виртуални машини, подобни на JRE. За разлика от JRE обаче, тези виртуални машини са предназначени за изпълнение на конкретна програма и не съдържат излишни за тази програма класове и функции. Поради това те са още по-малки в понятията на заемано дисково пространство.

Важно е да се знае, че в действителност съществуват няколко реализации на Java. Оригиналната виртуална машина за Java, разработена от Sun Microsystems и трансформирана в проект с отворен код по-късно, е известна под името „HotSpot“. Наред с нея съществува и виртуална машина, разработена от IBM и направена по-късно проект с отворен код. Тази машина е известна като „OpenJ9“ или „Semeru“. И двете виртуални машини се придържат към едни и същи спецификации и са до голяма степен еквивалентни. В момента се разработват и други виртуални машини, които ще се придържат към спецификацията на Java. Една от тях е GraalVM, която набляга на повишена производителност и възможности за компилиране на Java до машинен код.

JDK пакети на база HotSpot виртуална машина или OpenJ9 виртуална машина могат да бъдат изтеглени от <https://adoptium.net/> и <https://developer.ibm.com/languages/java/semeru-runtimes/downloads/>.

3. „Hello World“ приложение

Традиционна първа стъпка при изучаването на един език е съставянето на приложение, което отпечатва даден текст на екрана. По стара традиция този текст в учебниците и курсовете по програмиране често е „Hello, world!“. Ролята на „Hello, world“ приложението не е само да запознае потребителя с основните части на една програма на езика, който бива изучаван, но и да подsigури, че обучаемите знаят как да въведат и компилират всички примери, които ще бъдат дадени нататък в курса или учебника.

Както повечето програми, с които ще демонстрираме особеностите на Java в настоящия курс, програмата по-долу представлява конзолно приложение. Конзолните приложения нямат графичен потребителски интерфейс и работят в текстов режим на конзолата (Linux) или командния промпт (Windows).

```
1 package uk.co.lalev.javabook;  
2 public class Main {  
3     public static void main(String[] args) {  
4         System.out.println("Hello, world!");  
5     }  
6 }
```

Разпечатка 1.1. „Hello World“ приложение на Java

Програмата е цели 6 реда, като същинската работа се извършва на ред 4. **System.out.println("Hello, world!")** отпечатва текста „Hello, world!“ на конзолата.

Програмният код на Java се състои от последователности от подобни „инструкции“. Някои от тях инструктират виртуалната машина да извърши определени изчисления, а други извеждат стойности на конзолата или манипулират графичния потребителски интерфейс. Когато държим на прецизно и професионално изразяване, ние наричаме тези инструкции „**декларации**“, тъй като те декларират пред виртуалната машина какво трябва да бъде извършено, а тя на свой ред извършва указаните от декларациите действия. След всяка

декларация в Java трябва да се поставя точка и запетая, точно както е направено на ред 4.

Също както при естествения език, при който пишещият съставя последователност от изречения, за да опише дадена по-сложна мисъл, когато трябва да опишат някакво могостъпково изчисление, програмистите на Java формират последователности от декларации. Самата последователност от стъпки, описани чрез декларациите, се нарича „**алгоритъм**“, а конкретното описание на алгоритъма чрез декларациите на дадения език наричаме „**програма**“.

При разказите или романите, написани на естествени човешки езици, само изреченията не са достатъчни, за да структурираме добре мисълта си, поради което се нуждаем от параграфи, глави, точки и т.н. При програмните езици ситуацията е много подобна. Те разполагат с различни механизми за групиране и структуриране на декларациите в по-големи логически единици.

Една от най-важните единици за структуриране на програмния код е това, което в Java наричаме „**метод**“. Методите са начин за групиране на логически цялостна последователност от декларации под уникално име. Методите при нужда могат да се извикват множество пъти в програмата, като при всяко извикване се изпълнява последователността от декларации, включена в тях.

Така например в програма, която използва тригонометрия, може да съществува метод на име **sin**, който да обединява декларации, описващи действията, които са необходими за изчисляване на синус на даден ъгъл. На свой ред този метод може да бъде извикван многократно от метод, който се нуждае от изчисляване на синус, за да получи стойност на определен интеграл. Този пък метод на свой ред може да се използва многократно за изчисляване на прогнозна цена на финансови активи, което всъщност е предназначението на програмата.

Използването на методи за структуриране на програмния код е естествена идея с големи предимства. То не само елиминира нуждата от многократно писане на едни и същи последователности от декларации в програмата, но и позволява

лесно разпределение на работата, когато върху програмата работят повече програмисти. В контекста на горната хипотетична програма, един програмист може да работи върху разработването на метода **sin**, докато друг да работи върху метода за изчисление на интеграл, и т.н. В края на процеса всички програмисти ще обединят кода си и ще формират крайната програма.

В примерната програма на разпечатка 1.1 използваме готов метод, предоставен от Java. Това е методът **System.out.println**. Този метод отпечатва текст на конзолата. **System.out.println** обаче се нуждае от указание точно какво да отпечата. Указанията са дадени в скобите след името на функцията и всъщност се състоят от текста за опечатване, заграден в двойни кавички – "Hello, world".

Много методи се нуждаят от подобни „указания“. На професионален език наричаме тези указания „аргументи“ или понякога - „параметри“. В контекста на хипотетичния пример, даден по-горе, методът **sin** също трябва да има параметър и това е големината на самия ъгъл, за който ще се изчислява синус.

В примера на разпечатка 1.1. ние не само използваме готов метод, но и декларираме собствен метод. Това се случва на ред 3, който гласи:

```
public static void main(String[] args) {
```

Повечето от ключовите думи на този ред като **public** и **static** ще бъдат обяснени по-нататък. На този етап е важно само да се уточни, че името на метода е **main**. Всяка програма на Java трябва да има точно така дефиниран метод, тъй като по конвенция, виртуалната машина на Java започва изпълнението на програмата от него, като го извиква автоматично. Ето защо методът **main** се нарича „входна точка в програмата“ и в една програма може да има само един такъв метод.

Декларациите, които попадат в състава на един метод, се ограждат със фигурни скоби и се наричат „тяло на метода“. Тялото на метода **main** на нашата програма започва с фигурната скоба на ред 3 и завършва с фигурната скоба на ред

5. В случая тялото на метода се състои само от разгледаната вече декларация, разположена на ред 4.

В Java и другите програмни езици съществуват и по-големи единици за групиране на кода. Методи, които работят върху една конкретна същност от реалния свят, се групират в т.нар. "класове". Така например методите за показване, скриване и изчертаване на графичен прозорец могат да бъдат обединени в клас с име „Window”. Организацията на методите в класове позволява извършването на множество дълбоки оптимизации, които са обект на т.нар. „обектно-ориентирано програмиране“.

Java е силно обектно-ориентиран език, поради което не е възможно да имаме методи, които не са поставени в класове. Ето защо в нашата програма методът **main** е поставен в клас с име **Main**. Декларацията на класа започва на ред 2. Значението на ключовите думи преди името на класа също както при методите, ще бъде обяснено по-нататък. За да се обозначат методите, които попадат в даден клас, те се заграждат на свой ред във фигурни скоби. Така отварящата фигурната скоба на ред 2 обозначава „началото“ на класа **Main**, а фигурната скоба на ред 6 обозначава „края“ на класа. В случая класът **Main** има само един метод, но трябва да се знае, че в професионалните програми се използват по-сложни класове с множество методи.

Класовете се групират в още по-големи структурни единици, които представляват цели приложения или цялостни модули към такива приложения. Тези единици се наричат „пакети“. На първи ред на нашата програма ние имаме специалната декларация

package uk.co.lalev.javabook;

Тази декларация указва, че нашият клас ще бъде част от пакет с име **uk.co.lalev.javabook**.

За да се избегнат двусмислия в програмния код, Java изисква имената на пакетите да са уникални в рамките на една програма. По същия начин не може да има два класа с еднакви имена в рамките на един пакет. Изискването за уникални имена на пакетите е особено голям проблем в ситуациите, когато разработчиците споделят код помежду си. Може да се окаже например, че два или повече разработчици са кръстили своите пакети **javabook**. Включването и на двата пакета в една програма не може да се случи, без един от разработчиците да преименува своя пакет.

За да се избегнат такива неприятни ситуации, е прието имената на пакетите да включват като префикс Интернет домейна на компанията, която е разработила пакета, изписано в обратен ред. Авторът на настоящата книга притежава домейна **lalev.co.uk**, така че префиксира имената на пакетите си с **uk.co.lalev**. Така името на пакета на изложената програма не е просто **javabook**, а **uk.co.lalev.javabook**. Това име няма да съвпадне с нито едно друго име на пакет, произведен от друга компания или разработчик.

Читателите не трябва да се притесняват от това, че евентуално нямат собствен домейн. Техният код, разработен с обучителна цел, едва ли ще се споделя и включва в чужди програми, така че те могат да изберат всяко име, при условие че то не влиза в конфликт с някои от вградените в стандартната библиотеката на Java пакети.

4. Работа с **java**, **javac** и **jar** от командния ред

В наши дни професионалните приложения на Java се компилират с помощта на специални инструменти за построяване на проекти като Maven и Gradle. Последните често се управляват от среди за разработка на софтуер като IntelliJ IDEA и Eclipse.

Споменатите инструменти работят на по-високо ниво и спестяват на разработчика нуждата да извиква **javac**, **java** и **jar** на дневна база (всъщност

хиляди пъти на ден). Въпреки това, особено при внедряването на приложенията към крайните потребители, редовно се налага да се отстраняват проблеми „на ниско ниво“, които изискват познаването на работата на споменатите инструменти. Ето защо честният и пълен подход при запознаването с Java изисква тези инструменти да бъдат представени поне с техните основни опции, както е направено тук.

Сблъсквайки се със следващите няколко параграфа, много от начинаещи програмисти могат да бъдат обезкуражени. Те трябва да знаят, че ако използват среда за разработка, запознаването с **javac**, **java** и **jar** може да бъде отложено за по-удобно време.

Въпреки това е изключително полезно долните примери да бъдат „проиграни“ при възможност, тъй като те дават много добра основа за дискусия на организацията на кода при Java и начините, по които работи виртуалната машина – теми, към които настоящият курс ще се връща периодично.

4.1. Работа с **javac**

Компиляторът за Java компилира файлове с изходен код до файлове в байткод. Според спецификациите на езика, компилаторът изисква изходния код на всеки клас с модификатор за достъп **public** да бъде поставен в отделен файл с името на класа. Тоест клас с име **HelloWorld** трябва да бъде поместен във файл с име **HelloWorld.java**.

Въпреки че това не е изискване на самия компилатор, повечето инструменти за построяване на приложения в Java очакват ако класът е част от пакет (както винаги трябва да бъде в професионалните програми), файлът с неговия изходен код да бъде поставен в последователност от подпапки, които отговарят на името на пакета.

Така например Hello World приложението от предната подточка съдържа само един клас с име **Main**. Изходният код на този клас задължително трябва да

бъде поместен във файл с име **Main.java**. А тъй като класът **Main** е част от пакета **uk.co.lalev.javabook**, препоръчително е **Main.java** да бъде посавен в подпапка **uk/co/lalev/javabook**.

Долната последователност от команди демонстрира как се извършва компилацията на програмата от предната подточка под Windows.

```
1 c:\>mkdir project
2
3 c:\>cd project
4
5 c:\project>mkdir src\uk\co\lalev\javabook
6
7 c:\project>dir
8 Volume in drive C has no label.
9 Volume Serial Number is 9C9D-B8BA
10
11 Directory of C:\project
12
13 25.07.2021 г. 17:15 <DIR> .
14 25.07.2021 г. 17:15 <DIR> ..
15 25.07.2021 г. 17:15 <DIR> src
16 0 File(s) 0 bytes
17 3 Dir(s) 2 161 946 624 bytes free
18
19 c:\project>cd src\uk\co\lalev\javabook
20
21 c:\project\src\uk\co\lalev\javabook>copy con Main.java
22 package uk.co.lalev.javabook;
23 public class Main {
24     public static void main(String[] args) {
25         System.out.println("Hello, world!");
26     }
27 }
28 ^Z
29 1 file(s) copied.
30
31 c:\project\src\uk\co\lalev\javabook>notepad Main.java
32
33 c:\project\src\uk\co\lalev\javabook>cd \project
34
35 c:\project>javac src\uk\co\lalev\javabook\Main.java
36
37 c:\project>dir src\uk\co\lalev\javabook\*
38
39 ...
40
41 Directory of C:\project\src\uk\co\lalev\javabook
42
43 25.07.2021 г. 17:17 <DIR> .
44 25.07.2021 г. 17:17 <DIR> ..
45 25.07.2021 г. 17:17 436 Main.class
46 25.07.2021 г. 17:16 152 Main.java
47 2 File(s) 588 bytes
48 2 Dir(s) 2 161 946 624 bytes free
```

Разпечатка 1.2. Компилиране на „Hello World“ приложение под командния ред на Windows

Първата команда от ред 1 на разпечатка 1.2 създава основната папка на нашия проект. В случая потребителят е избрал папка с име **project**. На ред 3 потребителят създава подпапка **src**, която ще съдържа изходния код на нашата програма и едновременно с това създава нейните подпапки **uk/co/lalev/javabook**, които ще съдържат класовете от пакета **uk.co.lalev.javabook**. В конкретния случай този пакет има само един клас – класът **Main**.

Файлът **Main.java** се създава на редове 21-28 в папка **C:\project\src\uk\co\lalev\javabook**. Показаният начин е изключително екзотичен и е избран само защото това е най-ясният начин да се демонстрира в разпечатка какъв файл с какво съдържание бива създаден за целите на примера. Типично потребителите биха желали да използват интерактивни инструменти за създаването на файлове. Най-елементарният такъв инструмент е текстовият редактор **Notepad**, който е вграден във всички инсталации на Windows. Стартирането на **Notepad** е показано на ред 31 за тези читатели, които биха желали да пресъздадат примера. Такива читатели биха пропуснали редове 21-28 и биха написали съдържанието на **Hello World** приложението директно в **Notepad**.

Веднъж щом структурата на папките е създадена и файловете с изходните класове са по местата си, може да се премине към компилация на приложението. В примера тя се извършва на ред 35.

Както е видно от разпечатката, в този конкретен случай **javac.exe** приема за параметър едно име на файл, който съдържа файла с изходния код на класа за компилация. По нататък ще бъде демонстрирано, че **javac** може да приема за аргументи списък от имена на файлове, съдържащи изходен код.

Тъй като изходният код на класа **Main** е коректен и не предизвиква никакви грешки при компилация, резултатът от изпълнението на компилатора на Java е създаването на файла **Main.class**, който съдържа компилирания байткод на класа. Редове 37-48 на разпечатката демонстрират, че ако няма други указания, компилаторът на Java поставя **.class** файловете в същите папки, в които се намират съответстващите им **.java** файлове.

Смесването на **.java** и **.class** файловете при големите програми е лоша идея по много причини. Една от тях е, че **.java** файловете не са нужни за изпълнение на крайната програма. В много случаи дори е нежелателно те да се разпространяват към крайните клиенти от съображения, свързани с авторското право. Смесването тогава създава затруднения, свързани с извличането на **.class** файловете и оформянето им във вид, удобен за използване от клиентите.

За щастие **javac** поддържа опцията **-d**, която указва алтернативна папка, в която да бъдат поставени **.class** файловете, които се получават при компилация. По утвърдена сред програмистите конвенция името на тази папка е **target** или **out** и тя се намира на същото ниво като директорията **src**. Така **src** съдържа изходния код на приложението, а **target** евентуално се оказва попълнена с **.class** файловете, които представляват компилирания байткод на приложението, подходящ за разпространяване към крайните потребители.

Разпечатка 1.3. демонстрира компилация на приложението предната точка с използването на опция **-d**.

```
1 c:\>mkdir project
2
3 c:\>cd project
4
5 c:\project>mkdir src\uk\co\lalev\javabook
6
7 c:\project>cd src\uk\co\lalev\javabook
8
9 c:\project\src\uk\co\lalev\javabook>copy con Main.java
10 package uk.co.lalev.javabook;
11 public class Main {
12     public static void main(String[] args) {
13         System.out.println("Hello, world!");
14     }
15 }
16 ^Z
```

```

17      1 file(s) copied.
18
19 c:\project\src\uk\co\lalev\javabook>cd c:\project
20
21 c:\project>mkdir target
22
23 c:\project>javac -d target src\uk\co\lalev\javabook\Main.java
24
25 c:\project>dir src\uk\co\lalev\javabook\
26
27 ...
28
29 Directory of c:\project\src\uk\co\lalev\javabook
30
31 25.07.2021 г.  17:46    <DIR>          .
32 25.07.2021 г.  17:46    <DIR>          ..
33 25.07.2021 г.  17:46                  150 Main.java
34                1 File(s)                150 bytes
35                2 Dir(s)    2 161 868 800 bytes free
36
37 c:\project>dir target\uk\co\lalev\javabook
38
39 ...
40
41
42
43 Directory of c:\project\target\uk\co\lalev\javabook
44
45 25.07.2021 г.  17:47    <DIR>          .
46 25.07.2021 г.  17:47    <DIR>          ..
47 25.07.2021 г.  17:47                  436 Main.class
48                1 File(s)                436 bytes
49                2 Dir(s)    2 161 868 800 bytes free
50
51 c:\project>

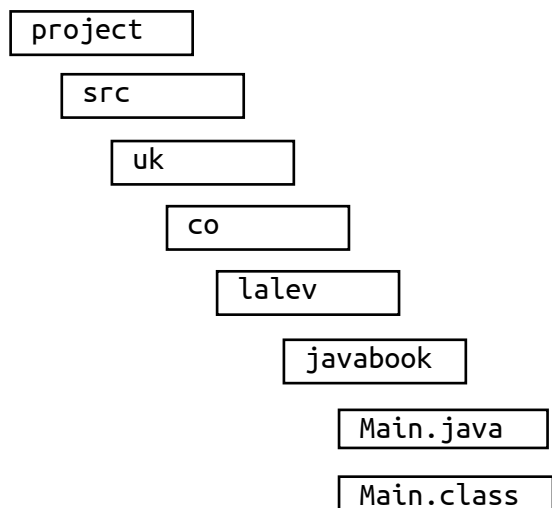
```

Разпечатка 1.3. Компилиране на „Hello World“ приложение под Windows с използване на опцията **-d**

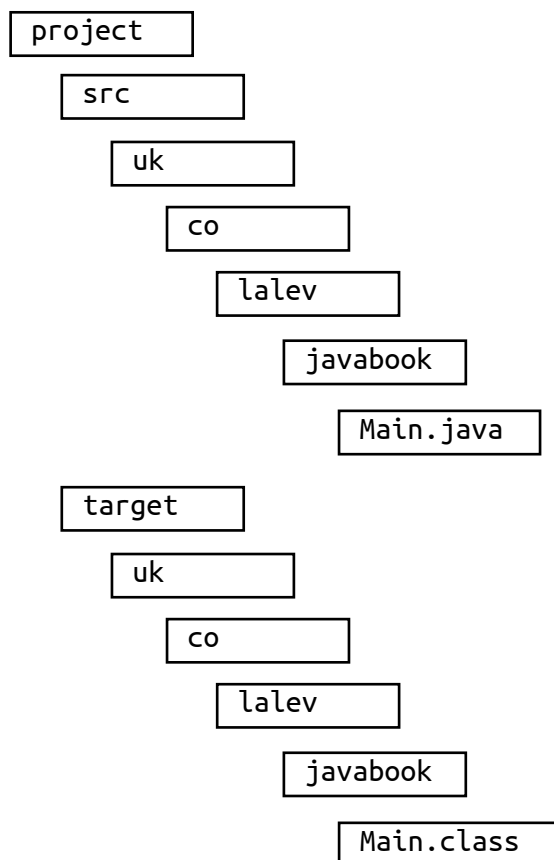
Действията, демонстрирани до ред 21 на разпечатка 1.3, са идентични с действията, демонстрирани на разпечатка 1.2. На тези първи редове потребителят създава подпапката **src** и в нея помества изходния код на програмата, който се състои от един клас, поставен в подпапки, отговарящи на името на пакета, който съдържа този клас. На ред 21 обаче, потребителят създава папка **target** като подпапка на **project**. Тази папка е на същото ниво като папката **src**.

Същинската компилация се извършва на ред 23. Този път потребителят е подал опцията **-d** с параметър **target** преди името на файла с изходния код. Резултатът от това е, че **javac** сега ще постави файла **Main.class** в подпапка **target**

вместо в същата папка, където се намира **Main.java**. Както е демонстрирано на редове 29-50, компилаторът ще създаде подпапките, определени от **package** декларациите на отделните класове. Така файлът **Main.class** всъщност ще се озове в папка **c:/project/target/uk/co/lalev/javabook**. Този факт е важен и е добре да бъде илюстриран с диаграма:



Фигура 1А. Използване на *javac* без опцията *-d*



Фигура 1Б. Използване на *javac* с опцията *-d* и посочване на папка *target*.

Друг важен параметър на **javac** е опцията **--class-path**, която алтернативно може да бъде посочена като **-cp** или **-classpath**. Тази опция е много важна, тъй като тя указва къде да бъдат търсени всички външни класове, използвани от текущо компилирания клас. Тъй като професионалните програми на Java се състоят от множество класове, те на практика не могат да бъдат компилирани без използването ѝ.

Ще илюстрираме проблема с малък професионален проект. Библиотеката *picocli* позволява създаването на Java програми, които работят от командния ред по начин, много по-добър на самите инструменти на Java. Този проект е много добър за нашия пример, тъй като по настоящем се състои само от два **.java** файла.

```
1 C:\>git clone https://github.com/remkop/picocli
2 Cloning into 'picocli'...
3 remote: Enumerating objects: 38919, done.
4 remote: Counting objects: 100% (178/178), done.
5 remote: Compressing objects: 100% (128/128), done.
```

```

6 remote: Total 38919 (delta 83), reused 110 (delta 43), pack-reused 38741
7 Receiving objects: 100% (38919/38919), 29.76 MiB | 7.98 MiB/s, done.
8 Resolving deltas: 100% (23112/23112), done.
9
10 C:\>cd picocli\src\main\java\picocli\
11
12 C:\picocli\src\main\java\picocli>dir *.java
13 Volume in drive C has no label.
14 Volume Serial Number is 9C9D-B8BA
15
16 Directory of C:\picocli\src\main\java\picocli
17
18 25.07.2021 г. 19:08          50 369 AutoComplete.java
19 25.07.2021 г. 19:08        1 206 784 CommandLine.java
20 25.07.2021 г. 19:08          1 298 package-info.java
21                3 File(s)        1 258 451 bytes
22                0 Dir(s)      2 053 246 976 bytes free
23
24 C:\picocli\src\main\java\picocli>mkdir c:\picocli\target
25
26 C:\picocli\src\main\java\picocli>javac -d c:\picocli\target
27 CommandLine.java
28
29 C:\picocli\src\main\java\picocli>javac -d c:\picocli\target
30 AutoComplete.java
31 AutoComplete.java:34: error: package picocli.CommandLine.Model does not
32 exist
33 import picocli.CommandLine.Model.PositionalParamSpec;
34                                     ^
35 C:\picocli\src\main\java\picocli>javac -d c:\picocli\target -cp
36 c:\picocli\target AutoComplete.java
37 C:\picocli\src\main\java\picocli>

```

Разпечатка 1.4. Компилиране на приложение с повече класове с използването на `--class-path`

На ред 1 на разпечатката, потребителят използва инструмента **git** (инсталира се отделно от Java) за да изтегли изходния код на **picocli**. Двата файла с изходния код се намират в подпапка на **src**, а именно **main\java\picocli**. Те са показани с командата на ред 12.

Първият от тях – **CommandLine.java** – е компилиран с командата на ред 26. Читателите следва да забележат, че компилаторът **javac** сега е извикан от същата папка, в която се намира и **.java** файла, докато в предните примери ние го извиквахме от папка нагоре по йерархията. Това е напълно допустимо.

Тъй като е използвана опцията **-d**, резултатният файл – **CommandLine.class** – ще бъде поставен в папката **target**. При това **javac** ще

създаде поредица от подпапки, които отговарят на името на пакета, към който се числи класът **CommandLine**. В случая разработчиците на **picocli** не са спазили гореописаните конвенции (дори професионалните разработчици понякога допускат грешки) и името на пакета е само **picocli** (тоест не съдържа името на домейн). Това означава, че резултатният клас ще се намира в папка **c:\picocli\target\picocli**.

До момента в примера не бе използвана опцията **--class-path**. За щастие класът **CommandLine.class** не използва други класове при извършване на функциите си, което позволява компилацията да се осъществи успешно. Класът **Autocomplete** обаче зависи от външен клас, а именно класа **CommandLine**. На ред 29 компилацията на **AutoComplete.java** завършва със серия от грешки, от които на разпечтатката е показана само една. Тя най-общо казва, че не може да бъде открит файл **CommandLine.class**, който е нужен за компилацията на **Autocomplete.java**. Програмистът коригира проблема с добавяне на опцията **-cp c:\picocli\target**. Това е достатъчно на компилатора да се ориентира в ситуацията и да потърси нужния **.class** файл в **c:\picocli\target\picocli** (за последната поддиректория във веригата компилаторът се ориентира по името на пакета, към който се числи **CommandLine**). Този път компилацията е успешна.

4.2 Работа с java

Командата **java** активира виртуалната машина. Основен параметър е името на класа, който е входна точка в програмата. Такъв клас трябва да има специално дефиниран метод **main**, както бе обяснено в предходната точка. Освен това виртуалната машина трябва да знае в кои папки тя може да намери **.class** файловете с байткода на всички класове, използвани в програмата. За целта **java** използва опцията **--class-path**, която е аналогична на тази при **javac**.

Връщайки се към т. 4.1. и предполагайки, че потребителят е компилирал HelloWorld приложението, отделяйки **.class** файловете в папка **c:\project\target**, приложението се стартира както е показано на следващата разпечатка.

1	
2	C:\>java -cp c:\project\target uk.co.lalev.javabook.Main
3	Hello, world!
4	
5	C:\>

Разпечатка 1.5. Стартиране на приложение на Java от командния ред

Опцията **-cp** указва “основната” директория с класовете на програмата, като **java** очаква байткода на всеки търсен клас да се намира в серия от подпапки, които отговарят на името на пакета, към който се числи съответния клас.

В случая **uk.co.lalev.javabook.Main** е напълно квалифицираното име на класа. Това име се получава от комбинирането на името на пакета и името на класа и вслучая указва в кой клас виртуалната машина трябва да търси входната точка на програмата.

Виртуалната машина ще търси файла с байткод на този клас в директорията **c:\project\target\uk\co\lalev\javabook**. Този път се получава като към директорията, подадена на **-cp** опцията, се добавят директориите, които отговарят на името на пакета, който съдържа класа **Main**.

4.3. Работа с jar

Читателите вероятно са забелязали, че стратегията да се генерира един по един файл с байткод за всеки клас води до наличие на множество **.class** файлове в програмата. Организацията на тези файлове по директории и разпространяването на тези директории към потребителите на програмата е доста по-сложно от това да се разпространява един файл. Именно поради тази причина Java използва специална архивационна програма, която създава архивни

файлове с байткода на класовете (тоест с **.class** файловете, получени при компилация). Тази програма се нарича **jar**.

От многото опции на програмата тук ще бъдат споменати само основните, а именно тези, които позволяват създаването на JAR архив. Надграждайки предния пример, следващата разпечатка показва как класовете от **c:\project\target** могат да бъдат архивирани и как резултатният JAR архив може да бъде подаден като параметър на **--class-path**.

```
1 C:\project\target>jar cvf c:\project\helloworld.jar .
2 added manifest
3 adding: uk/(in = 0) (out= 0)(stored 0%)
4 adding: uk/co/(in = 0) (out= 0)(stored 0%)
5 adding: uk/co/lalev/(in = 0) (out= 0)(stored 0%)
6 adding: uk/co/lalev/javabook/(in = 0) (out= 0)(stored 0%)
7 adding: uk/co/lalev/javabook/Main.class(in = 436) (out= 303)(deflated 30%)
8
9 C:\project\target>cd ..
10
11 C:\project>java -cp helloworld.jar uk.co.lalev.javabook.Main
12 Hello, world!
13
14 C:\project>
```

Разпечатка 1.6. Архивиране на .class файлове с jar

На разпечатката потребителят вече се е придвижил до папката **c:\project\target**, където се намират **.class** файловете на Hello World приложението. В тази папка той изпълнява командата **jar** с три параметъра **-c**, **-v** и **-f c:\project\helloworld.jar** (съкратено записани като **cvf**). Първият от тези параметри означава, че ще бъде създаден нов JAR архив, вторият означава, че за всеки файл, добавен към архива ще бъде изведен информационен ред и последната опция означава, че името на новия файл ще бъде **c:\project\helloworld.jar**.

Последният параметър обозначава директорията с **.class** файлове, която ще бъде поставена в архива. В случая този параметър е точка, което обозначава текущата директория. Читателите следва да забележат, че **jar** обхожда автоматично всички поддиректории на посочената директория в търсене на файлове, които да добави към архива.

Ред 11 показва, че навсякъде в **--class-path** аргумента, където може да се намира директория, може да се подава и местоположение на JAR архив. Читателите следва да сравнят този ред със съответния ред в предната разпечатка и да забележат разликите.

Модерните професионални приложения се състоят от множество компоненти и външни библиотеки. Разпространението на тези библиотеки става почти изключително под формата на JAR архиви, поради което не е въобще необичайно посочените чрез **-cp** JAR архиви да бъдат доста повече от един. В такъв случай имената на тези архиви се разделят с двоеточие.

5. Създаване на Hello World приложение с помощта на интегрирана среда за разработка

Читателите, проиграли примерите в предната точка, вероятно вече са наясно, че за разработката на Java програми трябва да има и по-ефективен начин, от писането им с обикновени текстови редактори и компилирането „на ръка“.

Интегрираните среди за разработка предлагат именно този по ефективен начин. Те могат да автоматизират извикването на инструменти като **java**, **javac**, **jar**, **jdb** и други, като в същото време предложат улеснения като синтактично оцветяване на ключовите думи, автоматично импортиране на класове и автоматично завършване на декларации. Ето защо те са безценен и задължителен инструмент от арсенала на професионалните програмисти.

Съществуват множество интегрирани среди, които поддържат разработка на Java. Такива например са IntelliJ IDEA, Eclipse, NetBeans, Visual Studio и Theia. Повечето от тях също като Java се разработват като проекти с отворен код, но има и такива, които са комерсиални продукти.

Давайки приоритет на решенията с отворен код, и отчитайки, че всяка една от изброените среди за разработка ще служи много успешно на начинаещите програмисти, ние избираме Eclipse за нашите демонстрации.

Eclipse е водещата среда с отворен код за разработка на Java. Полезно е да се знае, че Eclipse първоначално бе проект за създаване на такава среда. С времето общността около него нарастна и придоби сериозен авторитет, поради което днес на тази общност е делегирано управлението на други важни за Java проекти, което се осъществява чрез специално учредена фондация - Eclipse Foundation.

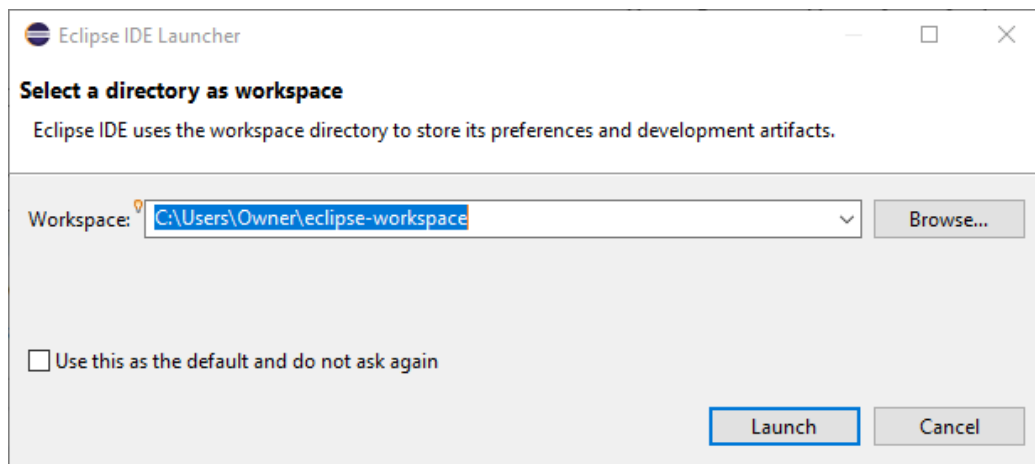
Тъй като името Eclipse се използва за обозначаване на най-различни проекти, доста често средата за разработка се нарича „Eclipse IDE“, а тъй като сайтът на Eclipse обслужва и други проекти, изтеглянето на средата за разработка не е съвсем праволинейно. Понастоящем изтеглянето на Eclipse IDE се извършва от адрес <https://www.eclipse.org/downloads/packages/installer> или от адрес <https://www.eclipse.org/downloads/packages/> в зависимост от това дали съответно става въпрос за изпълнима инсталационна програма или за архив от файлове.

Eclipse е модулна среда и може, с добавянето на подходящия набор от модули, да се превърне не само в среда за разработка Java, но и в среда за разработка на множество други езици, измежду които C и C++, PHP, Python и JavaScript. За улеснение на програмистите, инсталационната програма на Eclipse се предлага във варианти с инсталирани модули, подходящи за програмиране на конкретен език и конкретен контекст. Начинаещите програмисти се нуждаят от „Eclipse IDE for Java Developers“. Името на този пакет е сходно това на друг пакет - „Eclipse IDE for Java EE Developers“, който се използва от програмистите на Java, разработващи уеб приложения и друг съвършен софтуер с помощта на традиционните инструменти за тази цел, разработени от Oracle, Sun, IBM и други компании.

Повечето начинаещи биха желали да се придържат към „Eclipse IDE for Java Developers“ при разработване на конзолни приложения като представените в настоящата книга. Ако все пак читателите решат да програмират уеб приложения, те вероятно биха желали да използват за тази цел **Spring Boot**,

заедно със специално пакетирана и модифицирана среда за разработка на база Eclipse, която се разпространява от алтернативен сайт - <https://spring.io/tools>.

При стартирането си Eclipse извежда диалог със запитване за избор на работно пространство (фиг. 2).



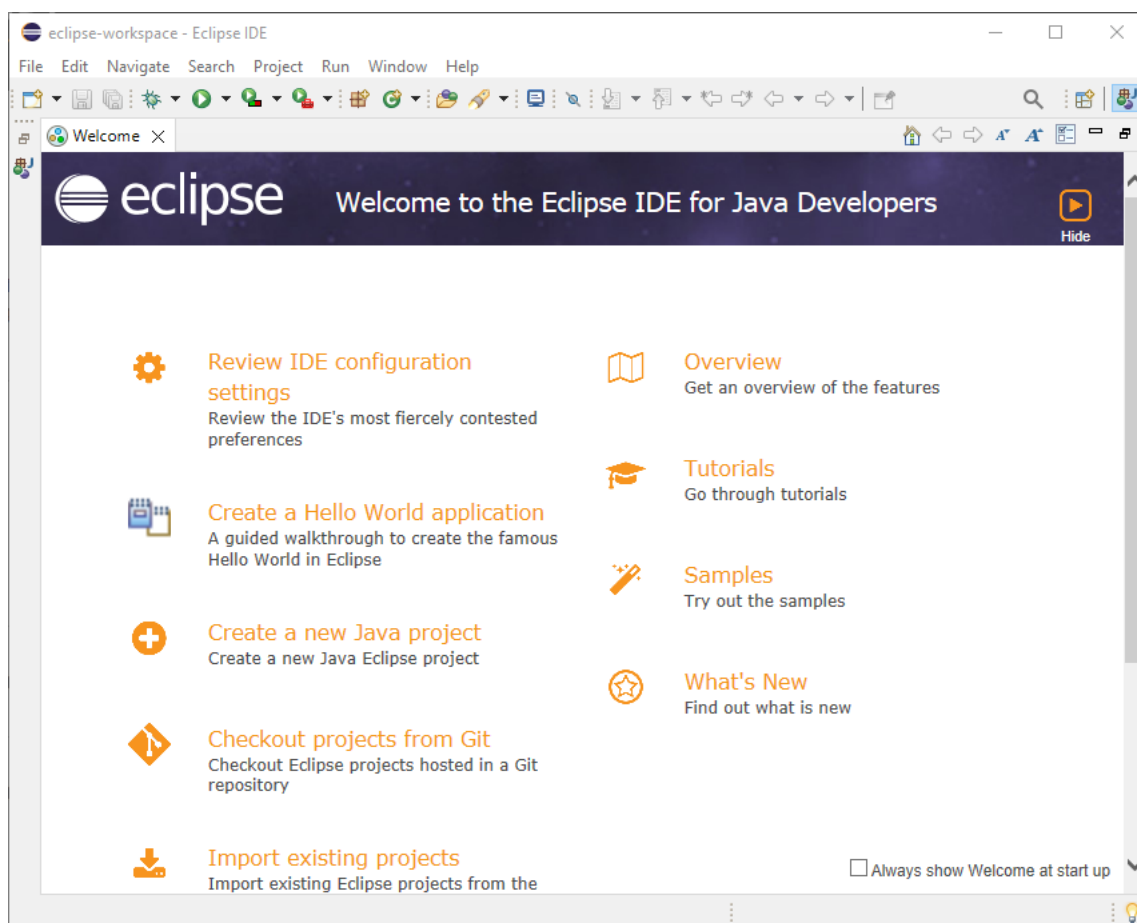
Фигура 2. Диалог за избор на работно пространство на Eclipse

Работното пространство представлява директория, в която Eclipse ще съхранява настройките си и ще създава подпапки за нови проекти. По подразбиране Eclipse ще използва за работно пространство специално създадена при инсталацията подпапка. Тази подпапка ще бъде разположена в домашната папка на потребителя.

Запитването е от полза за професионалните програмисти, които могат да желаят да имат няколко такива папки, обединяващи определени проекти със специфични настройки. Повечето начинаещи обаче вероятно биха желали да използват едно работно пространство. Ако запитването е твърде дразнещо, те могат да отметнат опцията **Use this as the default and do not ask again**. Тази опция води до запомняне на избора на папка и диалоговият прозорец няма да бъде показван при следващи стартирания на средата.

След като направи своя избор, разработчикът следва да натисне бутона **Launch**, което ще стартира средата с избраното работно пространство. При първо стартиране на средата от дадено работно пространство, Eclipse ще покаже Welcome прозорец, който дава кратък достъп до помощ за най-често

извършваните операции (фиг. 3). Този прозорец може да бъде затворен, като при нужда той може да бъде извикан от **Help/Welcome** в главното меню.



Фигура 3. Wellcome екран на Eclipse

Създаването на всякакъв вид проекти ред в Eclipse започва с извикването на **File/New/Java Project**, което води до показването на съветник (фиг. 4). В първия екран на този съветник се въвеждат някои от основните конфигурационни параметри на бъдещия проект:

Project name съдържа името на проекта. Това име ще бъде използвано на множество места за уникална идентификация на проекта, както и като име по подразбиране на папката, в която ще се съхраняват всички негови файлове.

Секцията **JRE** (името е запазено вероятно по исторически причини) всъщност позволява на потребителя да укаже **JDK**, който ще се използва за компилиране и стартиране на програмата. В тази секция има три основни опции. Опцията **Use an execution environment JRE** всъщност ще използва този JDK

пакет, който е използван за стартирането на самата среда. Ако не са правени други специални настройки на системата и Eclipse, това означава, че ще се използва вграденото и инсталирано заедно с Eclipse копие на JDK.

Използването на вградения JDK е напълно подходящо за начинаещи програмисти. Ако все пак е нужно да се използва отделно инсталиран вариант на JDK, за целта може да се използват втората или третата опция. Тези опции позволяват да се избере една инсталация на JDK от предварително формиран и конфигуриран списък с JDK инсталации на системата.

The screenshot shows the 'New Java Project' dialog box in the Eclipse IDE. The title bar reads 'New Java Project'. Inside, the section 'Create a Java Project' has the instruction 'Enter a project name.' Below this is a text field for 'Project name:'. A checked checkbox 'Use default location' is followed by a 'Location:' text field containing 'C:\Users\Owner\eclipse-workspace' and a 'Browse...' button. The 'JRE' section has three radio buttons: 'Use an execution environment JRE:' (selected), 'Use a project specific JRE:', and 'Use default JRE 'jre' and workspace compiler preferences'. The first option has a dropdown menu showing 'JavaSE-17'. The second option has a dropdown menu showing 'jre'. A link 'Configure JREs...' is to the right. The 'Project layout' section has two radio buttons: 'Use project folder as root for sources and class files' and 'Create separate folders for sources and class files' (selected). A link 'Configure default...' is to the right. The 'Working sets' section has an unchecked checkbox 'Add project to working sets', a 'New...' button, and a 'Working sets:' dropdown menu with a 'Select...' button. The 'Module' section has a checked checkbox 'Create module-info.java file', a 'Module name:' text field, and a checked checkbox 'Generate comments'. At the bottom, there is a help icon (?), and buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

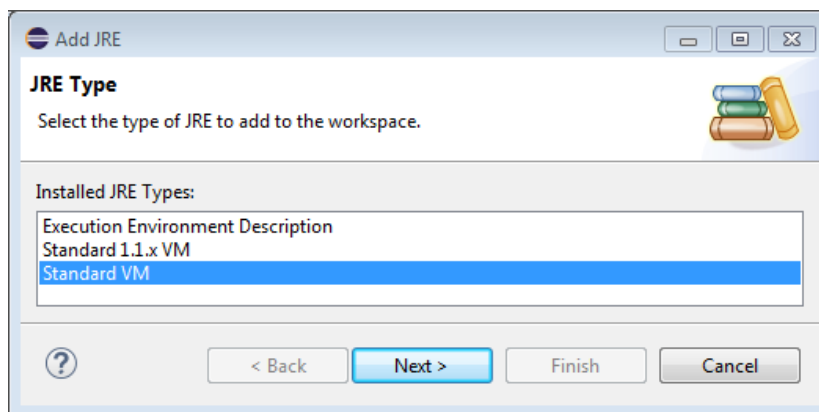
Фигура 4. Съветник за създаване на нов проект в Eclipse – начална страница

Този списък се управлява от връзката **Configure JREs...** в същата група. При щракване върху нея се отваря допълнителен диалог за конфигурация. Този диалог (фиг. 5) съдържа списъка, който винаги ще съдържа поне едно копие на JDK – това, което се използва за стартирането на самата среда. Eclipse дава името **jre** на това копие. То може да бъде разпознато и по това, че теговото местоположение се намира в подпапката **.p2** на домашната папка на потребителя.

Фигура 5. Списък на инсталираните копия на JDK, известни на Eclipse

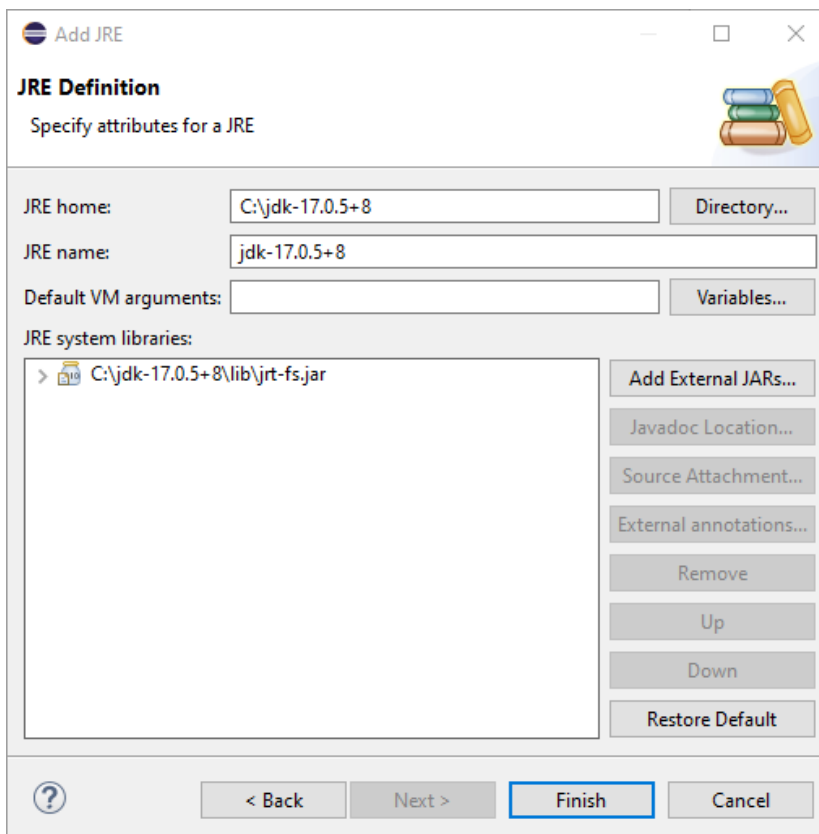
Добавянето на допълнителна инсталация на JDK в списъка става чрез бутона **Add...** При неговото натискане се отваря нов съветник за добавяне на JDK в няколко стъпки.

На първа стъпка (фиг. 6) потребителят трябва да избере типа на JDK. От нея следва да се избере **Standard VM**.



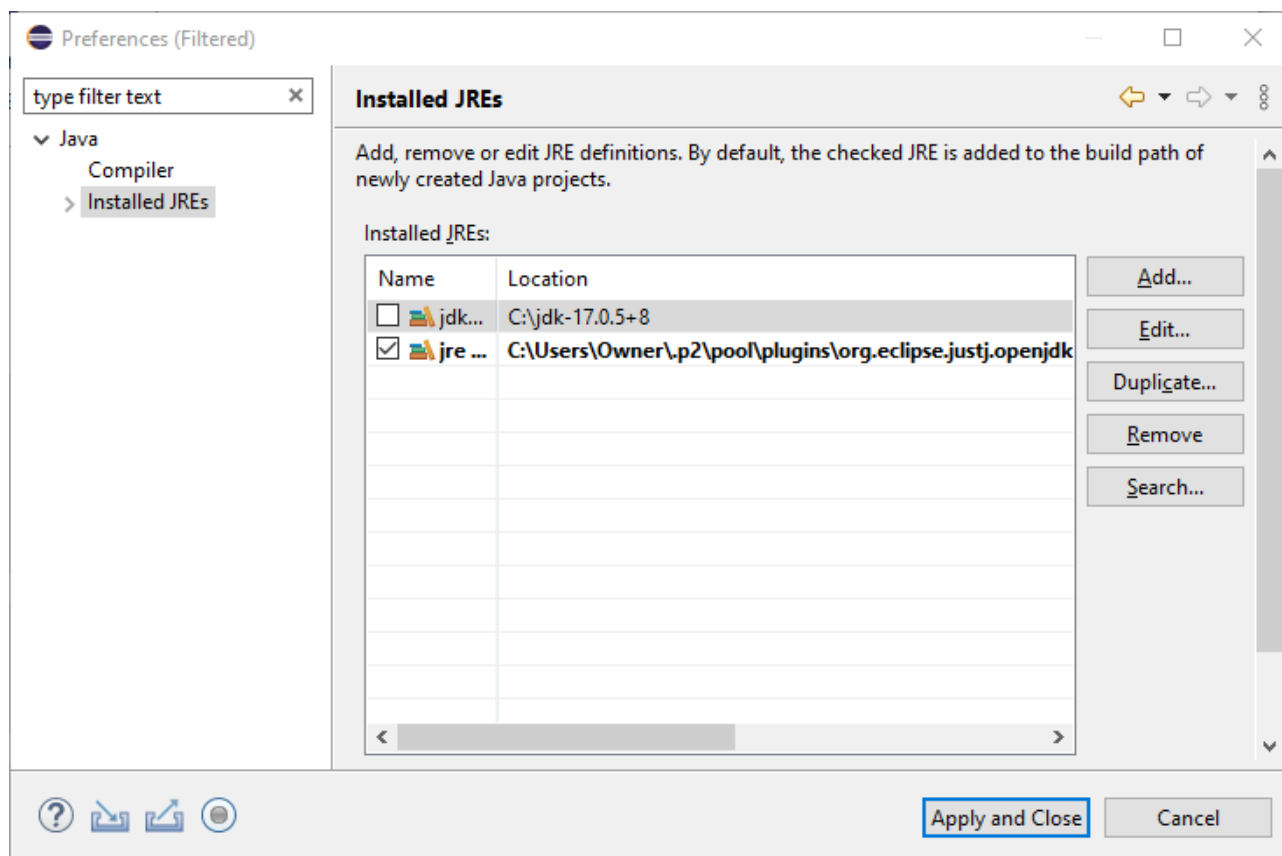
Фигура 6. Избор на тип на виртуалната машина

На втората стъпка (фиг. 7) потребителят трябва да укаже директорията, в която е инсталиран добавения JDK (Опция **JRE Home**). След избор на директорията, Eclipse ще попълни другите по-важни опции, включително името, под което добавения JDK ще влезе в списъка на средата (Опция **JRE name**). Стойностите по подразбиране за тези други опции са приемливи и трябва да се променят само ако има сериозна причина за това (например увеличаване на лимита на паметта, използвана от Java приложенията и т.н.)



Фигура 7. Указване на директорията, в която е инсталиран JDK

След като новото копие на JDK е добавено към списъка (фиг. 8) потребителят може да укаже, че то трябва да се използва по подразбиране за всички проекти в работното пространство. За целта той трябва да отбележи отметката пред името на нововъведеното JDK.



Фигура 8. Избор на JDK по подразбиране

Редактирането на JDK по подразбиране в списъка приключва с натискането на бутон **Apply and Close**.

Връщайки се към опциите в групата **JRE** на съветника за нов проект (фиг. 4), ще отбележим че втората и третата опция от групата са свързани точно с използването на JDK по подразбиране или обратното – използването на специфично JDK. Втората опция – **Use project specific JRE** - позволява специално за новия проект да бъде избрано копие на JDK от списъка. Третата опция – **Use default JRE** – позволява за текущия проект да се използва това JDK,

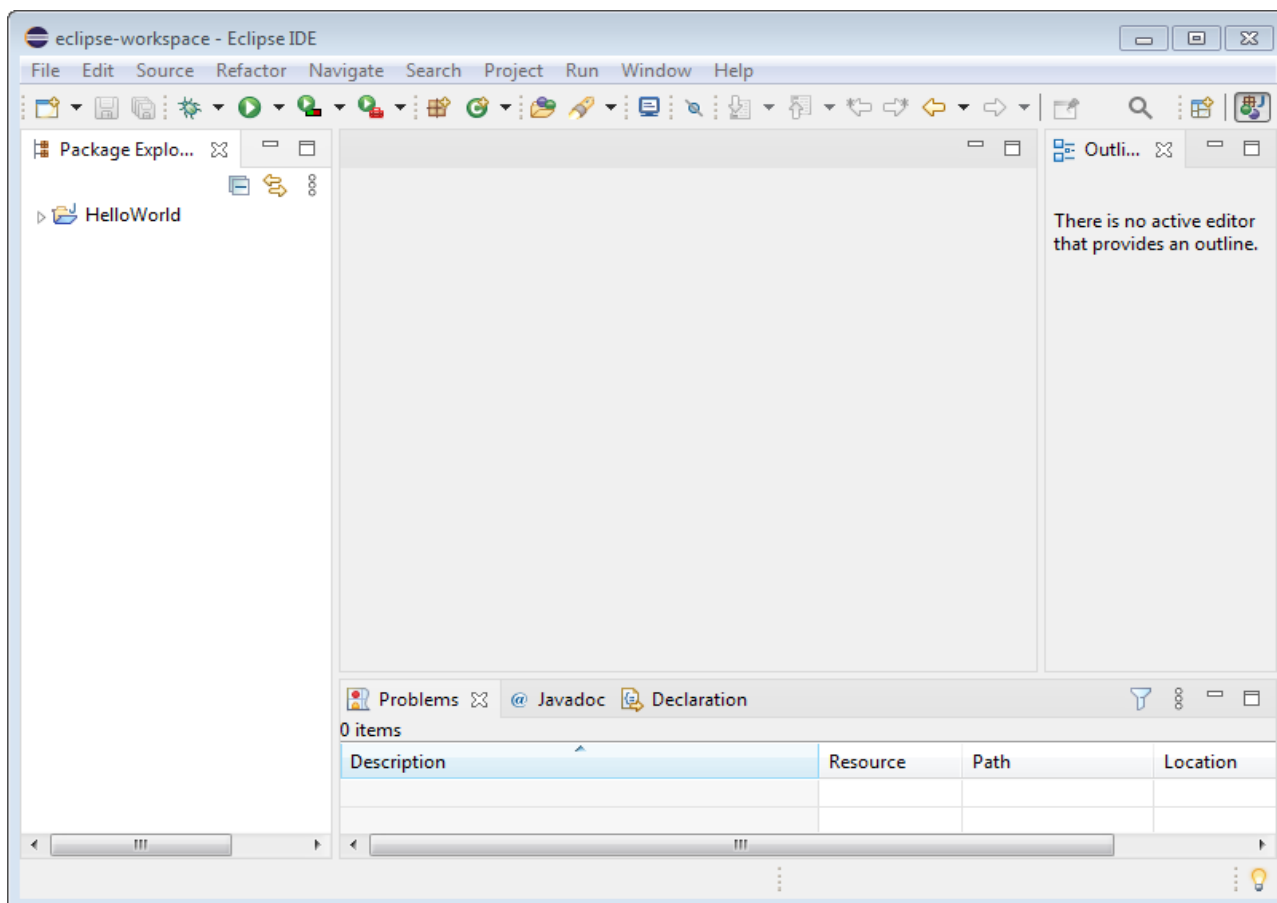
което в списъка е определено като JDK по подразбиране за цялото работно пространство.

В конкретния случай, за да създаде **HelloWorld** приложението и да избегне конфигурацията на модулите (които ще бъдат разгледани по-нататък), читателят може да даде за име на проекта **helloworld** и да избере от група JRE избор по свое усмотрение. Настройката на други опции не е необходима и след извършване на тези действия, читателят може да премине към втората стъпка на съветника чрез натискане на бутона **Next** или да натисне бутона **Finish**.

Втората стъпка на съветника (която се отваря само ако в предната е натиснат бутона **Next**) съдържа настройки за вградената система за построяване на приложения на Eclipse. Стойностите по подразбиране за тези настройки са напълно подходящи за нашето приложение и повечето учебни програми от настоящата книга, така че читателят спокойно може да натисне бутона **Finish** и да приключи съветника.

След създаване на новия проект той ще бъде автоматично отворен (фиг. 9). Работната област на Eclipse, като повечето подобни среди, е организирана като една централна област, която се използва за редактиране на програмния код. Встрани от тази основна област се намират разгъваеми панели.

При стартиране на нов проект пакетът **Package Explorer** се появява в разгъната форма и показва логическата структура на проекта. Този панел може да се използва за навигация до подпапката с кода на приложението. За целта програмистът трябва да щракне два пъти с мишката върху името на приложението в панела.



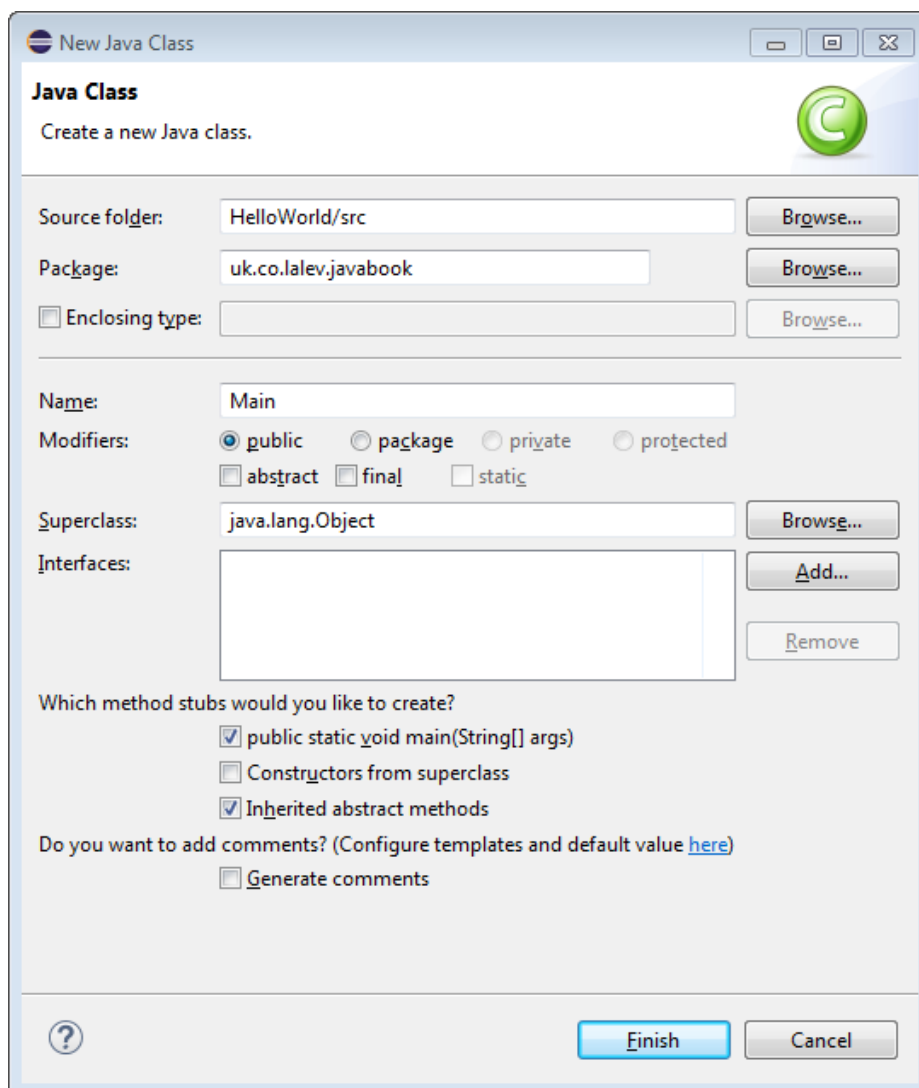
Фигура 9. Главен екран на Eclipse с новосъздаден проект

За да въведе кода на приложението от точка 3, програмистът трябва да създаде клас **Main**. За целта той може да извика **File/New/Class** от главното меню, което ще отвори нов диалогов прозорец за въвеждане на параметрите на класа (фиг. 10).

Диалоговият прозорец за създаване на нов клас съдържа няколко важни полета. Полето **Package** позволява въвеждането на име на пакет, в който ще бъде създаден новия клас. Ако пакетът не съществува, Eclipse ще го създаде автоматично. За създаване на **HelloWorld** приложението, програмистът следва да въведе стойност **uk.co.lalev.javabook** в това поле

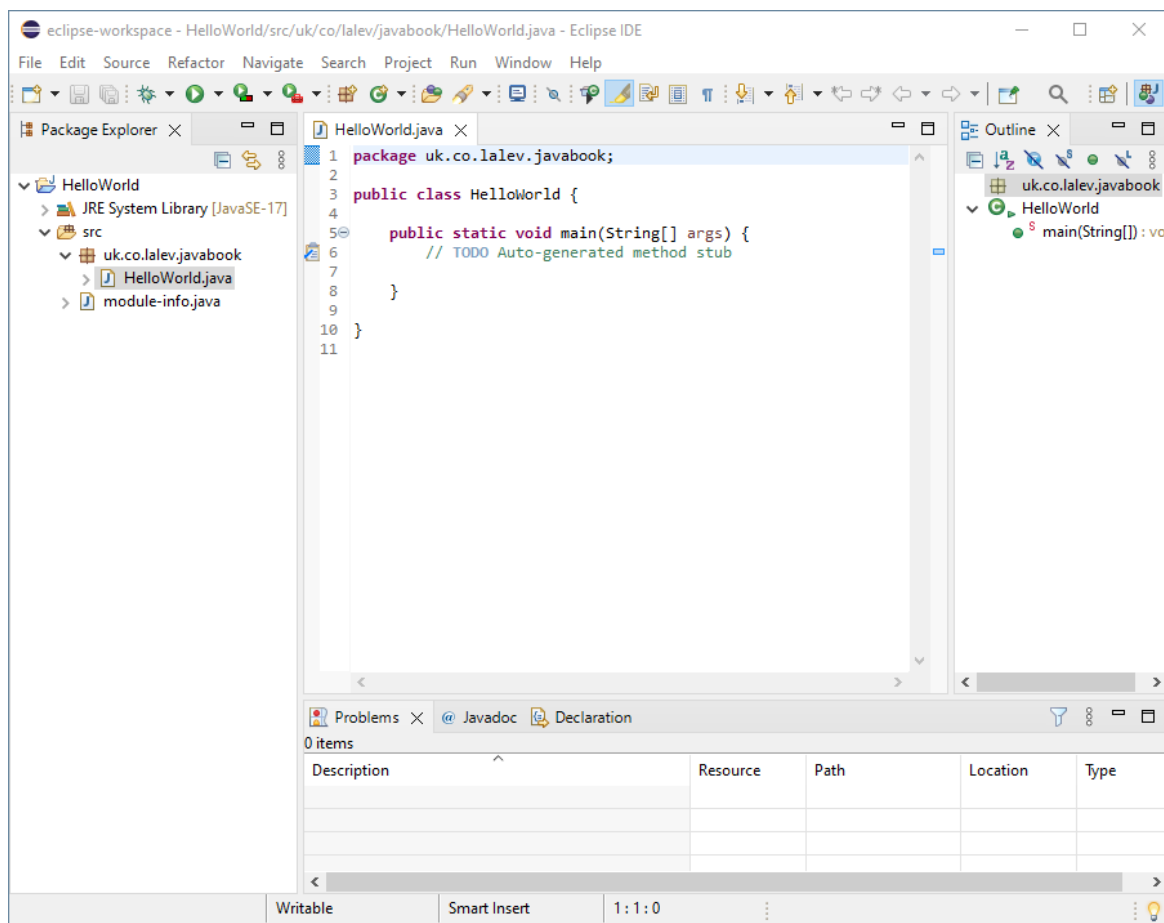
Полето **Name** съдържа името на новия клас и за създаването на примерното приложение, потребителят трябва да въведе **Main**.

Ако потребителят иска да инструктира средата да създаде автоматично и метод - входна точка в програмата, той трябва да отметне опцията **public static void main(String[] args)** в секция **Which method stubs would you like to create?**



Фиг. 10. Създаване на нов клас към съществуващ проект на Eclipse

След въвеждането на тези параметри, читателят може да приключи диалога с натискане на бутона **Finish**, след което средата ще създаде файл **Main.java**, който съдържа скелета на бъдещето HelloWorld приложение (фиг. 11).




Фигура 11. Проект на Eclipse с отворен клас

Така генерираното приложение съдържа почти всички декларации на HelloWorld приложението от т. 3. Единствената корекция, която трябва да бъде извършена от читателя, е замяната на ред

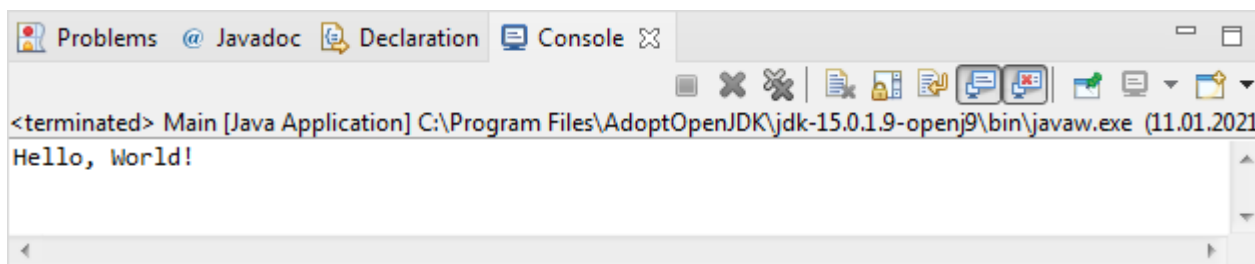
```
//TODO Auto generated method stub
```

с ред

```
System.out.println("Hello, World!");
```

След нанасянето на тази корекция, приложението следва да бъде записано. За целта потребителят може да извика **File/Save All**, да натисне бутона  или да натисне клавишната комбинация **Ctrl+Shift+S**.

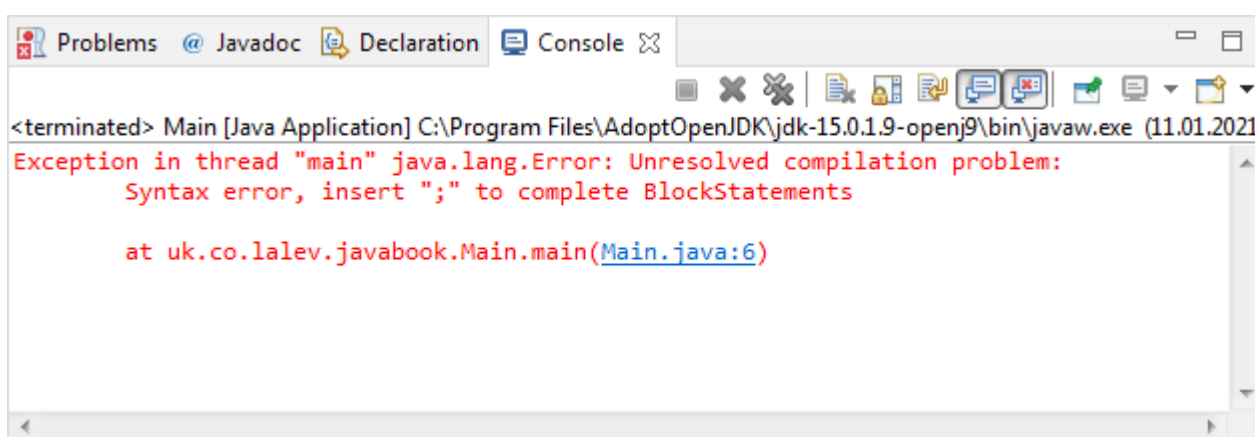
Компилацията и изпълнението на програмата могат да се извършат и на една стъпка с инструментите на средата. За да компилира и стартира програмата, програмистът може да извика **Run/Run** от главното меню или да натисне **Ctrl+F11**. При успешно изпълнение в долната част на екрана ще бъде разгънат панел **Console** (фиг. 12). Той ще показва изхода от програмата, който в случая е текста „Hello, World!“



Фигура 12. Панел Console с резултатите от изпълнението на Hello, World приложението

Ако възникнат грешки при компилацията, средата за разработка ще ги покаже в друг разгънат панел – панелът **Problems**. Фиг. 13 демонстрира как този панел показва съобщението за грешка от компилатора на Java, породено от изпуснатата точка и запетая след декларацията

```
System.out.println("Hello, World!")
```



Фигура 13. Панел Console с информация за грешка при компилация на програма

От съобщението може лесно да бъде установено, че компилаторът е прекратил компилацията на ред 6 на **Main.java**. Реалната грешка, предизвикала проблема, може да се намира на този или по-преден ред.

Глава 2. Елементарни типове, променливи и литерали

1. Елементарни типове в Java

Java е силно типизиран език, което означава, че всички променливи и всички данни, с които се работи в програмата, трябва да имат ясно разписан тип.

Типовете в Java се делят на две големи групи – **референтни** и **елементарни**. Референтните типове са тясно свързани с обектно-ориентираното програмиране и пълните детайли за тях ще могат да бъдат изложени едва в следващите глави. Елементарните типове, които често се наричат неформално и „**примитивни**“ са общо осем и са подробно описани в долната таблица:

Тип	Предназначение	Размер в паметта	Граници на възможните стойности
byte	съхраняване на цели числа	8 бита ²	от -128 до +127
short	съхраняване на цели числа	16 бита	от -32768 до +32767
int	съхраняване на цели числа	32 бита	от -2147483648 до +2147483647
long	съхраняване на цели числа	64 бита	от -9223372036854775808 до +9223372036854775807
float	съхраняване на дробни числа	32 бита	6-7 значещи цифри
double	съхраняване на дробни числа	64 бита	15-16 значещи цифри
boolean	съхраняване на логически стойности	8 бита	само две възможни стойности - true и false
char	съхраняване на символи*	16 бита	от 0 до 65535*

Таблица 1. Елементарни типове в Java

Типовете **byte**, **short**, **int** и **long**, се използват за съхраняване на цели числа. Типовете **double** и **float** пък се използват за съхраняване на дробни числа. Причината за наличието на четири типа за цели числа и два типа за дробни е свързана с ефективно използване на паметта на компютъра. Поради различни хардуерни и исторически причини, „естественият“ тип променлива за цели числа в Java е типът **int**. Създателите на Java са имали предвид той да бъде използван както за малки числа, така и за големи числа. Променливите от тип **int** са

² В компютърните системи измерваме обема на данните в понятията на битове. Ако даден тип заема 16 бита в паметта, това означава че в отделения му обем от паметта се поместват 16 двоични цифри.

достатъчно големи, за да съхранят повечето от целите числа, които възникват в практиката. В същото време 32-битовите и по-новите от тях процесори могат да извършват аритметични операции с такива числа на само една стъпка, така че изчисленията с тях се извършват много бързо.

Променливите за цели числа с по-малка дължина в паметта от **int**, (тоест **byte** и **short**) се използват основно когато трябва да се съхранят *наистина големи* съвкупности от сравнително *малки* числа. Така например, ако става въпрос за една, десет и дори 100 или 1000 променливи, изборът на тип няма голямо отражение върху използваната памет от програмата. В такъв случай би следвало да изберем за тип на променливите **int**. Но ако решим например да съхраним няколко милиона числа в паметта, е добре да преценим дали се наистина се нуждаем от **int** или естеството на данните позволява да направим променливите, които ще съхраняват тези числа, по-малки и по този начин да спестим памет.

Променливите от тип **long** заемат 64 бита в паметта и се използват основно когато числата, с които се работи, не могат да бъдат поместени в променлива от тип **int**.

„Естественият“ тип, предназначен за съхраняване на дробни числа в Java, е 64-битовият **double**. Подобно на **short** и **byte**, 32-битовият **float** се използва за икономия на памет, когато говорим за голям брой променливи стойности, които не биха загубили точност при съхранението им в по-малкия тип.

Броят битове, които се заделят в паметта за съхранение на променлива от елементарен тип, е фиксиран и не може да се променя. Този брой често се нарича „**дължина**“ и определя максималната и минималната числова стойност, която може да бъде поместена в дадения тип. При целочислените типове определянето на максималните и минималните стойности на база дължината в битове е сравнително праволинейно. Всички максимални и минимални стойности за тези типове са показани в предходната таблица.

Определянето на максималните и минимални стойности за типовете **float** и **double** е по сложно, тъй като техните стойности се съхраняват в компютъра

като две отделни полета. Това е същата идея, стояща зад добре познатия от математиката и физиката „научен“ или „експоненциален запис“. При този запис големи числа като 123000000000 или 0,000000123 могат да се запишат много компактно съответно като $123 * 10^9$ и $123 * 10^{-9}$. Читателят е приканен да забележи, че ако трябва да запомним тези числа, ние трябва да запомним само 123 и 9 (респективно 123 и -9). Точно това прави и Java - за всяка променлива от тип **double** и **float**, Java съхранява вътрешно две *цели* числа. Първото число съдържа всички цифри на оригиналното число до мястото, откъдето нататък има само нули (т.нар. „**значещи цифри**“). Второто число указва в каква посока трябва да се „придвижи“ десетичната запетая на първото число, за да получим съхраненото дробно число. Реализацията на тази идея в компютрите и програмните езици често се нарича „**аритметика с плаваща запетая**“.

Поради очертания начин на запомняне на информацията, **float** и **double** могат да се използват за съхраняване и на много големи и много малки числа. Поради по-малкия си размер обаче, **float** позволява съхраняването на числа с до 6-7 значещи цифри, докато **double** позволява съхраняването на числа с до 15-16 значещи цифри. Когато значещите цифри на дадена стойност са повече от лимита, стойността ще бъде съхранена в паметта в *закръглен* вид (вж. фиг. 1).

Фигура 1. Значещи цифри при числата с плаваща запетая

Типът **boolean** е създаден за съхраняването на логическите стойности **true** и **false**. Тези стойности възникват изключително често като резултат от различни логически проверки и сравнения, които Java изпълнява по заръка на програмиста. Те се използват и за описание на данни, които имат само две състояния. Въпреки че в паметта на компютъра променливите от този тип се съхраняват, разбира се,

като числа, в Java **boolean** не е числов тип. Това означава, че съдържанието на променливи от този тип *не може* да се интерпретира в програмата като числа.

Типът **char** в Java се използва сравнително рядко и е предназначен за съхраняване на единичен символ. Особеностите на този тип са много и са в голяма степен обвързани с еволюцията на начините, по които се съхраняват и обработват символи в компютъра. С неговите 16 бита дължина **char** може да съхранява числа от 0 до 65535, почти всяко от които по подразбиране се интерпретира като един символ от **кодовата таблица** Unicode.

Unicode представлява стандартна таблица за съпоставяне на символи и букви с числа, което позволява компютрите да поставят в паметта си и да обработват текстове (вж. табл. 2). Така например, за да представи латинската буква малко 'a', съгласно Unicode, Java поставя в променливите от символен и текстов тип числото 97.

символ (лат.)	число	символ (кир.)	число	символ	число
a	97	а	1072	#	35
b	98	б	1073	\$	36
c	99	в	1074	%	37

Таблица 2. Извадка от кодовата таблица Unicode

Най-важното нещо, което начинаещите програмисти трябва да знаят за типа **char**, е може би това, че той е вече остарял начин за представяне на една буква. По времето на писането на първите спецификации на езика Java (около 1994-1995 година) таблицата Unicode съдържа много по-малко от 65536 символа. Тези символи са достатъчни, за да се представят повечето азбуки, използвани по света, и към тогавашния момент изглежда, че 16-те бита на **char** винаги ще бъдат достатъчни за съхраняване на номерата на символите от Unicode. С течение на 25-те години от появата на Java, към Unicode се добавиха много не-буквени знаци (например такива, които представляват ноти, математически символи и пр.). Днес

броят на символите в Unicode надвишава 140 хиляди. Символите с уникални номера след 65535 обаче не могат да се поставят в променливи от тип **char**!

За да не променя дължината на типа и евентуално да „повреди“ компилацията на съществуващ програмен код, Java избра да не променя дължината на **char**. Вместо това промени бяха нанесени в съответстващия му референтен тип **Character** (референтните типове ще бъдат разгледани по-нататък в изложението). Този референтен тип е стандартният начин, който се използва за съхраняване на *единични* букви и символи в нови програми на Java.

2. Променливи

Компютърните програми обработват данни. Те получават определени данни за вход и извеждат определени данни като изход. В процеса на обработка обаче те трябва да съхраняват множество междинни резултати. За тази цел в компютъра съществува специално обособен хардуерен компонент. Този компонент е оперативната памет на компютъра.

Променливите в програмите на Java позволяват удобен и безопасен достъп до определени „клетки“ от оперативната памет и имат уникално име и тип. Името може да бъде всеки валиден идентификатор в Java, а типът показва какъв вид данни (числа, текст, дати и т.н.) могат да се съхраняват в дадената променлива.

За да се използва една променлива в дадена програма, тя трябва да се декларира, като в декларацията се посочват два задължителни и евентуално – един опционален елемент. Тези елементи са:

- **тип на променливата;**
- **име на променливата;**
- **началната стойност на променливата /незадължителен елемент/;**

Долният пример (вж. разп. 1) демонстрира декларирането на променливи от различни типове със и без задаването на начална стойност.

```

1 package uk.co.lalev.javabook;
2
3 public class VariableDeclarationDemo {
4     public static void main(String[] args) {
5         int length;
6         double weight;
7         String description;
8         boolean selected;
9
10        int ageOfRetirement = 65;
11        double pi = 3.1415926535897;
12        String name = "Петър";
13        boolean employed = true;
14    }
15 }

```

Разпечатка 1. Пример за дефиниране на променливи в Java

За да се дефинира променлива, първо се посочва нейния тип, след което се посочва нейното име. Тези два елемента са достатъчни, за да бъде това валидна декларация.

В примера на ред 5 се дефинира променлива от тип **int**, която се нарича **length**. На ред 6 се дефинира променлива от тип **double**, която се нарича **weight**. На редове 7 и 8 програмистът дефинира съответно променлива от тип **String** с име **description** и променлива от тип **boolean** с име **selected**.

Много променливи ще получат стойността си от външен за програмата източник, като споменатите диск, мрежа, конзола и т.н. В момента на писане на програмата дори не можем да предвидим точно каква ще бъде тази стойност, когато програмата се стартира с конкретни данни (това е точно причината по която наричаме тази клетка от паметта с името „променлива“).

Има обаче и такива променливи, които съгласно логиката на изчисленията ще получат ясно фиксирана начална стойност, известна още по времето на писане на програмата. За такива променливи началната стойност може да се зададе още при декларирането им.

Задаването на начална стойност на променлива се извършва като след името на променливата се добавя знак за равенство, следван от стойността. Това е демонстрирано на редове 10 до 13 от разп. 1.

Веднъж получили стойност, променливите могат да участват в различни изчисления. Пример за такова изчисление е дадено в долната рапечатка (разп. 2).

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4     public static void main(String[] args) {
5         double pi=3.1415926535897;
6         double radius=2.0;
7         double area;
8         area=pi*radius*radius;
9         System.out.println("Лицето на кръг с радиус "+radius+" е "+area);
10    }
11 }
```

Разпечатка 2. Променливи в Java

Тъй като знакът `*` в Java представлява оператор за умножение, **`pi*radius*radius`** е инструкция към Java да умножи текущата стойност на променливата **`pi`** два пъти по стойността на променливата **`radius`**. С други думи, това е инструкцията за изчисляване на формулата за лице на кръг (πr^2).

Инструкциите за това изчисление са част от по-голямата декларация **`area=pi*radius*radius;`** Тя включва оператора `=`, който в Java се интерпретира малко по-различно от приетото в математиката. **Знакът за равенство в Java е инструкция да се извършат изчисленията, които се намират отдясно на знака и резултатът от тях да се постави в променливата, която се намира отляво.** Тоест, инструкцията е да се извърши изчислението πr^2 и резултатът да се постави в променливата **`area`**. Тъй като знакът за равенство дава инструкции каква стойност да се постави в променливата вляво, често на професионален език той се нарича „**оператор за присвояване (на стойност)**“.

Резултатът от изпълнението на програмата е извеждането на текста:

```
Лицето на кръг с радиус 2.0 е 12.5663706143588
```

3. Литерали

Литералите са онези фиксирани стойности, които стоят в кода на програмите на Java. Те много често се използват за задаване на начални стойности на променливите, но могат да участват и в най-различни изрази. Ние използвахме вече литералите множество пъти. Всъщност, което е свидетелство за тяхната важна роля в програмите по принцип, ние използвахме литерали във всяка една програма до момента. Следващият програмен сегмент представлява още една илюстрация, като този път той е специално коментиран и всеки литерал е специално обозначен:

```
12  int radius = 5;           // 5 е литерал
13  double pi = 3.1415;      // 3.1415 е литерал
14  double circumference = pi*radius;
15  /*
16   * Долният ред има два литерала. "Обиколката на кръг с радиус" и " е ".
17   */
18  System.out.println("Обиколката на кръг с радиус "+radius+" е " +
19                      circumference);
20
21  int height=6; // 6 е литерал
22  /*
23   * Долният ред също има два литерала. Това са двете цифри 2.
24   */
25  double surfaceAreaCylinder = 2*pi*radius*radius + 2*pi*radius*height;
```

Разпечатка 3. Литерали в Java

Литералите в Java също имат типове, които кореспондират с елементарните типове.

Всяко *цяло* число, изписано в програма на Java, се интерпретира като литерал от тип **int**. По същия начин всяко *дробно* число³ (като например 3.14) се интерпретира като литерал от тип **double**.

Както бе споменато, **int** и **double** са основните числови типове за променливи и литерали в Java, докато останалите типове следва да се използват при много специфични нужди, очертани в предните точки. Ето защо в Java няма литерали от тип **byte** и **short**. При поставяне на стойности в променливи от тези

³ В Java и по принцип в повечето езици за програмиране използваме само десетични дробни. Изрази като 1/2, 1/4 и т.н. се интерпретират като деление, което в крайна сметка произвежда резултат, който отговаря на стойността на дробта, но е записан в паметта на компютъра като десетична дроб (т.е. 0.5, 0.25 и т.н.).

типове се използват литерали от тип **int** (специфичните особености на подобни операции са описани в следващата точка).

В Java обаче има литерали от тип **float** и **long**. Литерали от тип **float** се задават със суфикс **f** или **F**. Литералите от тип **long** се обозначават със суфикс **L** или **l**. Това е демонстрирано в следващия програмен сегмент:

```
7  int radius = 5;           // 5 е литерал от тип int
8  float pi = 3.14F;         // 3.14F е литерал от тип float
9  float e = 2.7182f;        // 2.7182 също е литерал от тип float
10 long l = 9876543210;      // !!! Грешка при компилация
11 long l1 = 9876543210L;    // 9876543210L е литерал от тип long
12 long l2 = 9876543210l;    // !!! Лош стил!
13                           // Малко "l" може да бъде прочетено като "1"
14                           // при използването на някои шрифтове!
```

Разпечатка 4 Литерали от тип *float* и *long* в Java

Ред 10 на разпечатка 4 илюстрира още една важна особеност на Java. Всички цели числа се интерпретират като литерали от тип **int**. Тези от тях, които са твърде големи, за да се съхранят като тип **int** и не са *експлицитно маркирани* от програмиста като тип **long**, генерират грешка при компилация на програмата.

Освен в десетична система, Java позволява въвеждане на целочислени литерали в двоична, осмична и шестнадесетична бройна система. За целта изписванията на числата се префиксират съответно с **0x**, **0** и **0b**, както това е демонстрирано на следващата разпечатка.

```
7  /* Всички литерали на редове 10-16 представляват числото 15, изписано в
8   * различни бройни системи
9   */
10 int a = 15;           // литерал от тип int, десетична бройна система
11 int b = 0xf;          // литерал от тип int, шестнадесетична бройна система
12 int c = 017;          // !!! Всички числа, изписвани с водеща нула се
13                       // интерпретират като зададени в осмична бройна
14                       // система! Иначе това отново е литерал от тип int.
15
16 int d = 0b1111;       //литерал от тип int
17
18 /* Литерали от тип long също могат да бъдат задавани в различни
19  * бройни системи.
20  */
21 long e = 0x7fffffffL; //литерал от тип long, изписан в
                       //шестнадесетична бройна система.
```

Разпечатка 5 Литерали от тип **int** и **long**, записани в различни бройни системи

В Java има и нецифрови литерали, които отговарят на нецифровите елементарни и на някои референтни типове. Следващият програмен фрагмент илюстрира използването на два вида литерали от „текстов“ характер.

6	char letter = 'a';	// 'a' е символен литерал от тип char
7	String name = "Иван";	// "Иван" е литерал от тип String
8	String whoops = 'z';	//Грешка!
9	char wow = "a";	//Грешка!

Разпечатка 6. Литерали от тип **char** и **String**

На ред 6 от разпечатка 6 програмистът декларира променлива от тип **char** и я инициализира с литерал от същия тип. Литералите от тип **char** могат да съдържат само един символ, който се изписва заграден в апострофи.

За разлика от тях литералите от тип **String** могат да съдържат произволно дълги последователности от символи (наричани още „**символни низове**“). Тези литерали се ограждат с двойни кавички, както е демонстрирано на ред 7.

Редове 8 и 9 демонстрират, че литералите от тип **String** и **char** не са взаимозаменяеми. На ред 8 програмистът се опитва да присвои на променлива от тип **String** литерал от тип **char**, което е недопустимо. На ред 9 се случва обратното. Въпреки че се състои от само една буква, за Java "a" е символен низ, тъй като е ограден в двойни кавички. На ред 9 програмистът прави опит да присвои този символен низ на променлива от тип **char**, което също не е позволено.

Литералите от тип **String** отговарят на променливи от тип **String**. Типът **String** има специален статус и всъщност е референтен тип, поради което ще го разгледаме подробно след като сме придобили повече интуиция за поведението на класовете в Java.

Два много специални литерала са **true** и **false**. Те представляват съответно логическите стойности истина и лъжа и са от тип **boolean**. Долният програмен сегмент демонстрира инициализацията на **boolean** променливи с тези литерали.

7	boolean a = true ;
8	boolean b = false ;
9	boolean c = TRUE ; //Грешка!

Както е демонстрирано на разпечатката, изписването на двата литерала (като всичко в Java) е чувствително към малки и големи букви. Правилното изписване на конкретните литерали е с малки букви.

Последен (но не по важност) е литералът **`null`**. В Java той се използва за да индикира, че дадена променлива от референтен тип в момента не сочи към обект в хийпа⁴ на програмата. Много методи в Java връщат **`null`** за да индикират, че в дадения случай липсва смислена стойност, която да върнат.

7	<code>String name = null;</code>
8	<code>int[] arr = null;</code>
9	<code>Object object = null;</code>

Разпечатка 8 Литерал **`null`**

Разпечатка 8 показва, че **`null`** може да се присвои на променливи от всякакви референтни типове. С разрастването на знанията ни за Java, в следващите глави ще можем да дадем повече смислени примери за употребата на **`null`**.

Завършваме точката с полезен съвет за начинаещите програмисти. Ако даден литерал се повтаря многократно в програмата, добра практика е той да се присвои на променлива веднъж и оттам нататък вместо литерала да се използва променливата. Кандидат за такъв литерал в икономически-ориентирана програма например би била стойността на Данък добавена стойност (ДДС). Числото „Пи“ се появява доста често в програмите, които извършват математически изчисления, поради което то също е добър кандидат да бъде заменено с променлива.

Причините да предпочитаме променливи вместо литерали за подобни повтарящи се стойности са две. На първо място те повишават четливостта на

⁴ Това в Java е регион от паметта на компютъра със специална организация, предназначен да помни по-голяма по обем и сложна по структура информация. Ние ще разгледаме някои особености на хийпа още в тази глава.

програмата. На второ място, когато евентуално стойността трябва да се промени, промяната ще стане на едно място вместо на множество места в програмата. Това е голямо предимство, тъй като промяната на един и същ литерал на множество места в програмата може да доведе до много неприятни грешки когато програмистът забрави да промени литерала на определена позиция.

Но въвеждането на променлива също носи своите рискове. Какво ако програмистът зададе нейната стойност правилно, но по-нататък в кода по погрешка промени стойността на променливата? Това е по-лесно да се случи, отколкото читателят си представя.

За да се избегнат подобни ситуации, Java има специален модификатор, който указва, че стойността на променливата се задава веднъж и няма да се променя по времето на съществуването на променливата. Този модификатор е **final**. **final** има много особености и се използва не само за променливи, поради което той ще бъде разгледан подробно в главите, посветени на класовете в Java. На този етап ще се задоволим с пример за използването на **final** към локални⁵ променливи.

Преди примера ще уточним, че такива променливи с фиксирана стойност реално ни позволяват да дадем уникално име на стойността, което да използваме в програмата като променлива. На професионален език ние често наричаме тези именувани стойности „**константи**“ по аналогия с математиката. Прието е имената на променливите, които съдържат константи, да се изписват с главни букви. Ако имената съдържат повече думи, прието е те да се разделят с подчертаваща черта.

⁵ Всички променливи, които ще разгледаме в първите глави на книгата, са на практика локални променливи, тъй като се намират вътре в даден метод. В Java има и променливи, които се декларират извън методи, но вътре в даден клас. Ние наричаме тези променливи „**полета**“ и ще ги разгледаме в главите, посветени на класовете и обектно-ориентираното програмиране.

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         final double PI = 22.0/7.0; //Тъй като pi не се променя никога,
7                                     //Java предлага готова final
8                                     //променлива - Math.PI
9                                     //която има по-прецизна стойност от
10                                    //горното грубо приближение.
11
12         final double BG_VAT = 0.2; // ДДС
13
14         PI=3.1415; //Грешка при компилация! Опитваме се да променим
15                  //променлива, маркирана като final.
16     }
17 }

```

Разпечатка 9. Използване на **final** променливи за съхраняване на константи

4. Понятие за референтни типове

По редица съображения от технически характер, Java разделя оперативната памет на две части, като достъпът до тези две части е организиран вътрешно по различен начин. Едната се нарича „**стек**“ (от англ. “stack”), докато другата се нарича „**хийп**” (от англ. “heap”).

Стекът е организиран по специален начин, което позволява на виртуалната машина на Java да поеме от програмиста напълно отговорността управление на жизнения цикъл на променливите. Когато изпълнението на програмата достигне до декларацията на дадена променлива, Java автоматично заделя нужната памет. Когато променливата вече не е нужна, Java маркира съответната част от паметта като свободна за други променливи.

Стекът е бърз, но по принцип **не може** да се използва така ефективно за съхраняване на данни, за които са верни едновременно две неща:

- дължината им в паметта е сравнително голяма;
- дължината не винаги може да се определи още по време на компилация на програмата;

Горното свойство на стека очертава разделителната линия между променливите от елементарен и референтен тип. Променливите от елементарен тип винаги имат фиксирана дължина, която освен това е милиони пъти по-малка от размера на паметта на модерните компютри. Поради тези две причини стойностите на променливите от елементарен тип се съхраняват директно в стека на програмата.

В компютърните програми обаче често се налага да се работи с данни, чийто обем не е ясен по време на компилация на програмата. Това работи срещу начините и механизмите на организация на стека. Много често подобни данни имат и значително по-голям обем, поради което е добра идея заделянето на памет да се отложи максимално - тоест не когато променливата бъде декларирана, а когато за първи път поставим стойност в нея.

Най-простият пример за такива данни са *символните низове*, споменати по-горе. Според контекста на обработката, символните низове могат да се състоят както от един, така и от милиони символи. Поради тази причина за тях не е практично да се поставят универсални ограничения за дължина и съответно - максимум брой символи, каквито има например при целочислените типове. Когато тези низове идват от външен за програмата източник (текстов файл, потребителски вход и пр.), ние не можем да знаем по време на компилация на програмата колко място ще е нужно за съхранението им в паметта.

За да реши проблема, Java предлага друг модел на съхранение, който поражда категорията на референтните типове. При референтните типове данните се съхраняват в хийпа на програмата, като място в хийпа на програмата се заделя до някаква степен ръчно от програмиста, който разписва инструкции за определяне на размера на входните данни и заделяне на съответния обем оперативна памет за съхраняването им. Съчетан с някои изключително добри алгоритми за управление на структури от данни, разработени през първите 50 години от развитие на теоретичната информатика и вградени в Java, хийпът на

програмата позволява да имаме данни, чийто размер расте и намалява плавно според нуждите на програмата.

Променливите от референтен тип съдържат препратка към мястото в паметта (по-конкретно – хийпа на програмата) в което се съхраняват реалните данни. Самата препратка има фиксирана дължина и малък размер, поради което нейната стойност се съхранява в стека.

Това означава, че всъщност стойностите на *всички* типове променливи се съхраняват в стека на програмата, но при променливите от елементарен тип стойността на променливата се поставя директно в стека, докато при променливите от референтен тип в стека се поставя препратка към местоположението на истинските данни.

Работата с референтни типове има много особености, които са неочаквани за начинаещите програмисти. Така например присвояването на една променлива от референтен тип на друга променлива от същия тип всъщност води до присвояването само на препратката. Така двете променливи в крайна сметка водят до един и същ обект в паметта. Променянето на обекта чрез едната променлива води веднага до променянето на стойността и на другата, тъй като и двете променливи всъщност са препратки към един и същ обект в паметта!

5. Автоматични превръщания между елементарни типове

До момента разгледахме много типове променливи и данните, които се съхраняват в тях, но избягахме старателно въпроса какво се случва, ако програмистът се опита да присвои на променлива от един тип данни от друг тип или се опита например да извърши събиране върху стойности от два различни типа. Оказва се, че нуждата от подобни операции възниква доста често в компютърните програми по най-различни обективни причини. Ето защо, за да улеснят програмистите, повечето езици за програмиране поддържат автоматичното превръщане на данни от един тип в друг. Тази операция на

професионален език се нарича още „преобразуване“ или „конверсия“ на типове⁶.

Преобразуването на типове в Java се управлява от три основни принципа:

- **Java проверява за валидността на превръщането по време на компилация;**
- **Java не позволява превръщания, при които могат да се загубят данни, без програмистът специално да е указал, че превръщането трябва да се извърши;**
- **Независимо от евентуални указания на програмиста, Java няма да позволи превръщания между несъвместими типове;**

Съществуват обаче много изключения, които са объркващи за начинаещите програмисти.

Java позволява превръщания както между елементарни, така и между референтни типове. В настоящата точка ще разгледаме само превръщанията между *елементарни типове* в контекста на *присвояването* на стойности на променливите.

Независимо от контекста, в който се извършват, **Java различава разширяващи и стесняващи превръщания.**

Разширяващите превръщания в повечето случаи не водят до загуба на информация. В редките екстремни случаи, когато те водят до загуба, тя се

⁶ Докато повечето езици за програмиране поддържат преобразуване на типове, езиците се различават съществено относно това колко „либерални“ са правилата, които управляват това преобразуване. По-либерални езици като JavaScript ще се опитат да превръщат автоматично почти всички видове данни в друг. Проблемът с либералната интерпретация на типовете се корени в това, че поощрява писането на опасен код, който може да съдържа грешки. Java се числи към групата на „стриктно-типизираните“ езици. Системата на типове на Java е така организирана, че максимален брой от грешките, свързани с типовете, ще бъдат открити още по време на компилация. Макар да не изглежда като голямо предимство „на хартия“, това доста често е разликата между улавянето на грешката още преди да стигне до клиента вместо улавянето и след като вече предадената и вкарана в експлоатация система внезапно спре или започне да поврежда реални данни!

изразява в закръгляване на стойността под въпрос, като разликите между закръглената и реалната стойност са пренебрежимо малки. Ето защо разширяващите превръщания са винаги позволени.

Стесняващите превръщания най-често водят до механично „отрязване“ на цифрите (по-точно двоичните такива или с други думи битовете) на по-голямото число, така че то да се помести в по-малката променлива. Това може да доведе до сериозна загуба на информация (например смяна на знака, драстична разлика в между оригиналното и резултатното число, нулиране на числото и т.н.). Поради тази причина стесняващите превръщания са позволени само при определени много специфични условия.

Долната таблица демонстрира вида на всяко превръщане между елементарните типове. Типът **boolean** е изключен от таблицата, тъй като той не е числов и неговите стойности не могат да се превръщат в числа. С други думи, превръщането от число в **boolean** е превръщане в *несъвместим тип* и Java няма да го позволи. Същото е вярно и за обратното превръщане – от **boolean** в число.

от \ към	byte	short	char	int	long	float	double
byte	---	P	C	P	P	P	P
short	C	---	C	P	P	P	P
char	C	C	---	P	P	P	P
int	C	C	C	---	P	P*	P
long	C	C	C	C	---	P*	P*
float	C	C	C	C	C	---	P*
double	C	C	C	C	C	C	---

Таблица 3. Разширяващи и стесняващи превръщания между елементарните типове в Java. Разширяващите превръщания са маркирани с „P“, докато стесняващите превръщания са маркирани със „C“.

Както табл. 3 показва, превръщането от 8-битовия *byte* към 32-битовия тип **int** е разширяващо и няма да загуби информация. Превръщането от 8 битовия **byte** към 16 битовия **char** обаче е стесняващо. Това се дължи на факта, че **char** приема само положителни стойности. Типът **byte** от друга страна може да има отрицателни стойности, които няма как да бъдат поместени в **char** без загуба на информация.

Четири от разширяващите превръщания, отбелязани в таблицата със звезда, се третират като разширяващи, въпреки че в някои редки случаи могат да загубят информация! Така например при превръщане от **int** във **float** може да се получи закръгляване на числото. При това единиците, десетиците и стотиците на числото, съхранени в променливата от тип **float**, могат да се разминават от същите позиции в оригиналното число от тип **int**. Тези разлики се наблюдават при близки до крайните положителни или отрицателни стойности, поради което дизайнерите на Java са приели да третират това превръщане като разширяващо. Още по-интересен е случаят с превръщането от **float** към **double**. Ако Java използва механизмите, вградени във виртуалната машина, изчислението ще се извърши малко по-бавно, но няма да бъде загубена точност. За целта методът или класът трябва да бъдат маркирани като **strictfp**. Ако обаче се използват хардуерните модули на процесора за работа с числа с плаваща запетая (поведението по подразбиране), Java не може да даде гаранции, че в някои редки случаи няма да се загуби точност.

Долната разпечатка илюстрира разширяващите превръщания:

```

1 package uk.co.lalev.javabook;
2
3 public class PrimitiveConversions {
4
5     public static void main(String[] args) {
6         int a = 5;
7         long b = 5;    // Разширяващо превръщане.
8                        // Литерал от тип int се превръща в long.
9         long c = a;    // Разширяващо превръщане. Стойност на
10                       // променлива от тип int се превръща в long.
11
12         float fpi = 3.1415f;
13         double dpi = fpi; // Разширяващо превръщане. Стойност на
14                           // променлива от тип float се
15                           // превръща в double.
16         double da = a;    // Разширяващо превръщане. Стойност на
17                           // променлива от тип int се превръща в double.
18
19         long test = 9223372036854775807L; // Макс. стойност за long.
20         double dtest = test;              // Разширяващо превръщане.
21                                           // long към double.
22         System.out.printf("%d\n", test);  // Между тази стойност
23                                           // и долната ще има малка разлика.
24         System.out.printf("%f\n", dtest);
25     }
26 }

```

Разпечатка 10. Разширяващите превръщания в Java са позволени при присвояване

Когато при *присвояване* се налага стесняващо превръщане от *литерал*, или *израз*, чиято стойност може да бъде определена по време на компилация, Java ще го разреши ако литералът или резултатът от изчислението на израза може да се помести в променливата без загуба на информация.

Долният пример (разп. 11) илюстрира най-простата ситуация, при която на променливата се присвоява директно литерал.

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         byte a = 5; //OK. Стесняващо превръщане на литерал от тип int към
7                     //byte.
8
9         byte b = 128; //Грешка. Стесняващо превръщане на литерал от тип int
10                      //към byte. Стойността на литерала е извън диапазона
11                      //на позволени стойности за byte.
12
13         char c = 65534; //OK. Стесняващо превръщане на литерал от тип int
14                       //към char.
15
16         char d = 65536; //Грешка. Стесняващо превръщане на литерал от тип
17                       //int към char. Стойността на литерала е извън
18                       //диапазона на позволени стойности за byte.
19
20         char e = -1; //Грешка. Стесняващо превръщане на литерал от тип
21                   //int към char. char не може да съхр. отриц. числа.
22
23         byte f = 'a'; //OK. Стесняващо превръщане на литерал от тип char
24                     //(Unicode код 97) към byte.
25
26         byte g = 'я'; //Грешка. Стесняващо превръщане на литерал от
27                     //тип char (Unicode код 1103) към byte;
28
29         float pi = 3.14; //!!! Грешка!
30     }
31 }

```

Разпечатка 11. Стесняващи превръщания в Java са позволени при присвояване на литерал от „по-широк“ тип, който обаче има стойност, която може да се помести без „отрязване“ в по-малкия тип.

При *присвояване* на стойност Java няма да позволи стесняващо превръщане на **double** литерал във **float** дори да изглежда, че литералът е в нужния диапазон. Това е демонстрирано на ред 29 на разпечатката и се дължи на редица особености при съхраняването на числа с плаваща запетая от тип **double** и **float** в паметта на компютъра.

Когато при присвояване на израз или променлива, чиито стойности не могат да се определят по време на компилация, се налага стесняващо превръщане, Java няма да го позволи без експлицитни инструкции на програмиста. Даването на такива указания се извършва чрез т.нар. „експлицитно преобразуване“ или още „експлицитен тайпкастинг“. Експлицитното преобразуване се задава като пред променливата,

стойността или израза, който трябва да се превърне в друг тип, се поставя името на желанния тип на резултата, ограден с кръгли скоби. Тази лексикална конструкция всъщност се третира като *оператор* в езика Java – „оператор за преобразуване“ или „тайпкастинг оператор“.

Разпечатка 12 демонстрира използването на тайпкастинг:

```
1 package uk.co.lalev.javabook;
2
3 public class Casting {
4
5     public static void main(String[] args) {
6         int a = 5;
7         double b = 3.14;
8         char d = 'a';
9         short e = -5;
10
11         byte f = (byte)a;    // OK
12         float g = (float)b;  // OK
13         short h = (short)d;  // OK
14         char i = (char)e;    // OK. Внимание! Стойността на i ще е
15                             // много различна от -5 !!!!
16         System.out.println((int)i); //65531
17
18         boolean j = (boolean)1; // Грешка при компилация.
19         String k = (String)i;   // Грешка при компилация.
20         boolean l = (boolean)a; // Грешка при компилация.
21
22         byte m = (byte)(a*1000); // OK. Стойността в m ще бъде много
23                                 // различна от 5000 !!!
24         System.out.println(m);  //-120
25         byte n = (byte)b;       //OK. Дробната част ще бъде отрязана!
26         System.out.println(n);  //3
27     }
28 }
```

Разпечатка 12. Тайпкастинг оператор

При извършването на тайпкастинг трябва да се помни, че Java няма да позволи тайпкастинга, когато двата типа са несъвместими. Такива несъвместими двойки от типове са демонстрирани на редове 18-20 на разпечатката. Независимо от указанията на програмиста Java няма да превърне числов литерал или променлива във **boolean** стойност, тъй като в системата от типове на Java това действие просто няма смисъл – **boolean** не е числов тип. По същия начин Java няма да превърне **boolean** в **String**.

Очертаното поведение е различно както от някои езици на ниско ниво като С, така и от някои езици на високо ниво, като JavaScript. Езикът С например би превърнал **boolean** число на база това, че стойностите **false** се представят с нула в паметта на компютъра. JavaScript и подобни езици биха превърнали логическата стойност **false** в символния низ *“false”*. Философията на Java обаче разчита на стриктното типизиране. Тоест тя настоява, че подобни превръщания трябва да се извършват от програмиста, за да се избегнат грешки при интерпретацията на програмния код от компилатора.

Ето защо, за да се извърши превръщане от един несъвместим тип към друг, целта в Java не са достатъчни само указанията, дадени под формата на тайпкастинг. Програмистът трябва да предостави и самия алгоритъм за превръщането, за да се погрижи за специфичната интерпретация на граничните случаи.

За да дадем по-ясна представа за такива гранични случаи например можем да споменем, че всъщност в класическите версии на С няма **boolean** тип. Вместо това за съхранение на стойностите **true** и **false** могат да се използват обикновени числови типове. Това дава интерпретацията 0 – **false** и 1 – **true**. Проблем обаче възниква когато различно от 0 и 1 число се подаде в израз, в който се очаква работа с логически стойности. Езикът С интерпретира такива положителни числа като **true**, което не е интуитивно и отваря възможности за грешки, които няма да бъдат открити по време на компилация на програмата. Така например променливата може да е попаднала в логическия израз по погрешка и компилаторът на С няма да предупреди програмиста за това.

Когато програмист на Java иска да извърши превръщане между несъвместими типове, като например **boolean** към **String**, той често ползва услугите на стандартната библиотека, която предлага множество методи и варианти за превръщане, които, разбира се, често са параметризируеми, за да позволят на програмиста лесно да зададе указания за превръщане на споменатите гранични случаи.

Глава 3. Оператори в Java

1. Аритметични оператори

Тъй като компютърните програми в крайна сметка са създадени за извършване на изчисления не е изненадващо, че аритметичните оператори в тях се срещат често и като цяло се записват по традиционно приетия в математиката начин. Събиране, изваждане, умножение и деление в Java се задават съответно със символите `+`, `-`, `*` и `/`.

За разлика от дурги езици, в Java няма оператори за повдигане на степен, извличане на корен и изчисляване на абсолютна стойност. За целта се използват методите на класа `java.lang.Math`, а именно `Math.pow()`, `Math.sqrt()` и `Math.abs()`.

Долният пример демонстрира използването на аритметични оператори и функции.

```
1 package uk.co.lalev.javabook;
2 /*
3  * Изчислява корените на квадратно уравнение от вида ax^2+bx+c=0
4  */
5 public class Main {
6     public static void main(String[] args) {
7         double a = 1;
8         double b = 3;
9         double c = -4;
10        double D = Math.pow(b,2)-4*a*c; //Дискриминанта
11        double x1 = (-b + Math.sqrt(D))/2*a; //Първи корен
12        double x2 = (-b - Math.sqrt(D))/2*a; //Втори корен
13    }
14 }
```

Разпечатка 1. Използване на аритметични оператори и методи на класа `Math` за изчисляване на корените на квадратно уравнение

Локиката зад изчисленията в тази малка програма всъщност не е особено релевантна, но дори и в рамките на толкова малко програмен код могат да бъдат демонстрирани важни особености. Една от тях възможността да се използват скоби за указване реда на операциите. Поредица от променливи и литерали,

съчетани с оператори и допълнени евентуално със скоби, често се нарича „израз“.

Важен оператор, който се използва в много алгоритми, е операторът за изчисляване на остатък или още “модул”. Той се отбелязва с `%` и дава остатък при целочислено деление на две числа, както е илюстрирано в долния програмен сегмент.

```
7  int a = 7;
8  int b = 3;
9  System.out.println(a%b); // Извежда 1, тъй като 3 се помещава в 7 два
10                             // пъти, а остатъкът е 1.
```

Разпечатка 2. Оператор за изчисляване на модул

2. Оператор за слепване на низове

Аритметичните оператори действат върху числени типове, но има едно важно изключение. Знакът за събиране може да се прилага и към символни низове. Когато поне един от операндите е символен низ, този знак се интерпретира като **оператор за слепване** (или още „оператор за конкатенация на низове“). При слепването символният низ отдясно на оператора се добавя към левия. Ако един от операндите не е символен низ, той се превръща към такъв.

Долният пример илюстрира този важен оператор:

```
1  package uk.co.lalev.inputoutput;
2
3  public class Main {
4      public static void main(String[] args) {
5          System.out.println("Здравей "+"свят!"); // Здравей, свят!
6          int value = 25;
7          System.out.println("Стойността е "+value); // Стойността е 25
8          String a = "A";
9          a = a+value;
10         System.out.println(a); // A25
11         System.out.println("Текст "+true+" "+3.14); // Текст true 3.14
12         System.out.println(true + false); //Грешка при компилация!
13     }
14 }
```

Разпечатка 3. Използване на оператор за слепване

Трябва да се отбележи, че докато знакът `+` може да се интерпретира като два различни оператора – за събиране и за слепване на низове, това е изключение. Много малко от операторите в Java се интерпретират двузначно според контекста. Така например операторите за умножение, деление и изваждане могат да се прилагат стриктно спрямо числови типове.

За да се интерпретира знакът за събиране като оператор за слепване, поне един от операндите трябва да бъде символен низ. Така например никой от операндите на ред 12 в предходната разпечатка не е символен низ, поради което Java интерпретира това като опит за събиране на логически стойности. Това не е позволено в Java и води до грешка при компилация.

В израза на ред 11 има два оператора, което поражда въпроса в какъв ред точно ще се изпълнят те. Ще разгледаме този въпрос по-подробно по-нататък в темата. Важно е да се знае обаче, че всички оператори работят върху две или върху една стойност. Повечето оператори са **бинарни**, т.е. работят върху две стойности – лява и дясна. **Унарните оператори** работят върху една стойност, който идва след тях. Когато един оператор бъде изпълнен от виртуалната машина на Java, той „трансформира“ операндите си в стойност, като тази стойност замества операндите и оператора в израза. Изчислението на израза по-нататък продължава точно по начина, който е познат на читателя от началния курс по алгебрата.

3. Оператор за присвояване

Читателите вече без съмнение вече познават оператора за присвояване от предходните теми. Той се изписва със знак за равенство (`=`) и взема два операнда. Левият операнд трябва да е променлива, която може да променя стойността си, докато десният операнд може да бъде литерал, променлива или израз, състоящ се от литерали и променливи, свързани с оператори. Долният пример отново демонстрира използването на оператор за присвояване:

6	<code>int a = 5; // Литерал 5 се присвоява на променлива a.</code>
7	<code>int b = a; // Стойността на променливата a</code>
8	<code>// се присвоява на променливата b</code>
9	<code>int c = ((a*5)+b)/6; // По-сложно присвояване</code>
10	<code>int d = c = 6; // Още по-сложно присвояване</code>

Разпечатка 4. Използване на оператор за слепване

Ред 9 от примера демонстрира, че операторът за присвояване се изчислява последен. По този начин изразът `((a*5)+b)/6` първо получава стойността си (5) и едва след това тази стойност се присвоява на променливата **c**. Както споменахме и по-рано, редът на изпълнение на операторите е важен, поради което ще получи доста по-подробно разглеждане в специална подточка на настоящата тема.

Ред 10 демонстрира едно сравнително неинтуитивно свойство на оператора за присвояване в Java. Освен че присвоява десния операнд на левия, операторът за присвояване дава резултат. Това му позволява да участва като дясна страна на всякакви други изрази, съставени от всякакви оператори, включително и други оператори за присвояване. Това свойство е използвано на ред 10, за да се установят променливите **d** и **c** на стойност 10.

В израза на ред 10 дясната страна на оператора за присвояване е **c=10**. Това на свой ред е израз с оператор за присвояване. Изпълнението на **c=10** ще установи стойността на променливата **c** на 10. Самият израз **c=10** ще бъде изчислен и ще получи стойност 10 – стойността на десния операнд на оператора за присвояване. След извършването на това действие, временното състояние на израза на ред 10 ще бъде **d=10**. Изчислението на оставащия оператор за присвояване ще доведе до установяване на променливата **d** на стойност 10.

Читателите на този етап може би си задават въпроса защо при наличие на два оператора за присвояване, изчислението протича в описания по-горе ред. Както ще обясним в детайли по-нататък, в изрази, съдържащи еднакви по приоритет оператори, изпълнението протича „отляво-надясно“.

4. Оператори за сравнение

Операторите за сравнение, често наричани още „**релационни оператори**“, са `>` (по-голямо), `<` (по-малко), `>=` (по-голямо или равно), `<=` (по-малко или равно), `==` (равно) и `!=` (различно)⁷. При срещане на релационен оператор, виртуалната машина проверява дали изразът отляво на оператора е в указаното отношение с израза от дясната страна. Ако това е така, Java връща за резултат от операцията **true** (който се интерпретира като „истина“/“вярно“). В противен случай Java ще върне **false** (който се интерпретира като „лъжа“/“невярно“).

Долният програмен сегмент илюстрира с примери свойствата на операторите за сравнение.

```
6  boolean result1 = 5==5;
7  System.out.println(result1); // true
8
9  boolean result2 = 5>5;
10 System.out.println(result2); // false
11
12 boolean result3 = 5>=5;
13 System.out.println(result3); // true
14
15 int r = 4;
16 double s = 3.14;
17 boolean result4 = (r/s)>2.15;
18 System.out.println(result4); // false
19
20 System.out.println("lalev"<"test"); //Грешка при компилация!
```

Разпечатка 5. Използване оператори за сравнение

Разпечатката акцентира върху множество важни моменти, свързани с операторите за сравнение.

На първо място, в Java (и повечето програмни езици) присвояването и логическият оператор за проверка на равенство се извършват от различни оператори. В Java единичният знак `=` е оператор за присвояване, чието действие вече наблюдавахме многократно в примерите досега. Два последователни знака

⁷ Към тези оператори често се причислява и операторът **instanceof**, който определя дали един обект е от даден тип. Тъй като този оператор е свързан с обектно-ориентираното програмиране, ние ще го разгледаме по-късно.

„равно“ (**==**) представляват оператор за сравнение, който проверява дали изразите от лявата и дясната страна на оператора са равни.

На ред 6 в разпечатката програмистът използва оператор за сравнение, за да накара виртуалната машина да извърши тривиалната проверка дали числото 5 е равно на себе си. Резултатът очаквано е **true**. Той се присвоява на променливата **result1** от тип **boolean**. В повечето реални програми резултатите от проверките, инициирани чрез използването на оператори за сравнение, въщност ще зависят от неизвестни по време на писането на програмата фактори, като входа от потребителя, съдържанието на файла и т.н. Това показва важността на тези оператори. Чрез тях програмата може да реагира по различен начин, когато получава различни входни данни.

Въпреки че представихме типа **boolean** по-рано, ние отложихме дискусията относно неговата употреба. Сега за пръв път го използваме в реална програма и можем да демонстрираме най-важната причина за неговото съществуване. В повечето случаи променливите от този тип се използват именно както е показано на разпечатката - за съхраняване на резултата от дадена проверка.

Проверките на редове 9 и 12 демонстрират разликите между „стриктните“ и „не-стриктните“ версии на операторите „по-голямо“ и „по-малко“. При използване на стриктна версия на оператора „по-малко“, например както е показано на ред 9 на разпечатката, лявата страна трябва да е стриктно по-малка от дясната, за да получим резултат **true**. Числото 5 не е стриктно по-малко от себе си, поради което тази конкретна проверка връща **false**. Не-стриктната версия на същия оператор, която наричаме „по-голямо или равно“ работи както стриктната с тази разлика, че когато двете страни се окажат равни, тя връща **true**. Това е демонстрирано на ред 12. Абсолютно по същия начин стоят нещата с операторите „по-голямо“ и „по-голямо или равно“.

Читателят трябва да забележи, че по традиция, не-стриктните версии на операторите „по-голямо“ и „по-малко“ се изписват с два отделни знака.

Използването на утвърдените в математиката знаци \geq и \leq няма да даде желанния резултат и ще произведе грешка при компилация на програмата.

Последна важна особеност, илюстрирана на разпечатка 3.2, е това, че в Java, за разлика от някои по-слабо типизирани езици, операторите „по-голямо“ и „по-малко“, както и техните не-стриктни версии, могат да се прилагат само върху числени типове и литерали (т.е. цели и дробни числа). Прилагането им върху променливи и литерали от тип **String** например, както е показано на ред 20, не е позволено. Операторите, които проверяват равенство и неравенство (`==` и `!=`) от друга страна могат да се прилагат (при определени специфични обстоятелства) и към други типове и особено по отношение на променливи и литерали от тип **boolean** и **char**.

5. Логически оператори

Операторите за сравнение могат да извършват едно сравнение. Много често в реалните програми проверките за настъпване на дадено условие са по-сложни и включват множество сравнения, които формират *логически израз*. Логическите оператори предоставят именно начини за свързване на сравненията в изрази. Те могат да се прилагат само върху променливи и литерали от тип *boolean*.

Операторът „логическо и“ се обозначава с `&&` (два последователни знака „амперсанд“). Неговото действие може да се опише със следната таблица.

операнд 1	операнд 2	резултат
false	false	false
true	false	false
false	true	false
true	true	true

Както показва таблицата, резултатът от оператора „логическо и“ е *true* само когато и двата логически израза отляво и отдясно на оператора са *true*. Долният пример илюстрира използването на оператора `&&` в реалистичен сценарий:

```

6  /* В реална програма стойностите на следващите две променливи ще
7  * идват от потребителски вход.
8  */
9  double vehicleMassKg = 1000.0; // маса на МПС в килограми
10 int seats = 9;                // брой места за пътници
11
12 boolean operatesUnderClassBLicense =
13     vehicleMassKg<3500 && seats<=8;    //false

```

Разпечатка 6. Използване оператор && („логическо и”)

Програмният сегмент на разпечатка 6 определя дали дадено моторно превозно средство, предназначено за превоз на пътници, може да се управлява със свидетелство за правоуправление клас „Б“. За целта трябва да са изпълнени и двете условия – масата на превозното средство да е под 3500 кг. и местата за пътници да не надвишават 8.

Редове 9 и 10 декларираат променливите *vehicleMassKg* и *seats*, които съдържат двете стойности, върху които ще извършваме проверката. Тези стойности са фиксирани в нашия пример, но в една реална програма биха идвали от външен за програмата източник, поради което нямаше да знаем техните стойности по време на писане на самата програма.

Същинската проверка се извършва на ред 13. При срещане на логически оператор, Java ще изчисли израза от лявата страна на оператора. Тази страна ще даде резултат **true** или **false**. В случая левият израз е **vehicleMassKg<3500**. Тъй като променливата **vehicleMass** на редове 12-13 има стойност 1000, резултатът от тази проверка (стойността на израза) ще бъде **true**.

Ако стойността на лявата страна е достатъчна за определяне на крайния резултат от логическия оператор, дясната страна няма да бъде изчислявана въобще. Така например **false** за лява страна на нашия оператор би означавало, че резултатът от неговото изпълнение е ясен. Горната таблица показва, че резултат **true** се получава само ако и двата операнда са **true**. Ако един от тях е **false**, общият резултат също ще е **false**, независимо каква е стойността на другия. В нашия случай обаче лявата страна е **true**, поради което ще бъде изчислен и израза отдясно на оператора.

Десният израз е `seats<=8`. Променливата `seats` обаче има стойност 9, поради което резултатът от проверката е **false**. След като получи лявата и дясната стойност, виртуалната машина най-накрая ще реши какъв ще бъде резултата от цялата операция. Съгласно горната таблица това ще е **false**, тъй като вторият операнд е **false**.

Интуитивната интерпретация на описаната сложна проверка е, че превозно средство с маса 1000 кг. и 9 места не може да се шофира със свидетелство от клас Б, тъй като броят седалки надвишава 8.

Операторът „логическо или“ се обозначава с две последователни вертикални черти `||`. Ефектите от изпълнението му са описани в следващата таблица:

операнд 1	операнд 2	резултат
false	false	false
true	false	true
false	true	true
true	true	true

С други думи, ако един от двата операнда е **true**, резултатът от изпълнението на оператора е също **true**. Долният пример демонстрира неговото действие:

6	<code>int age = 59;</code>
7	<code>int weightKg = 100;</code>
8	<code>boolean highRiskCovidPatient = weightKg>=90 age>=60; //true</code>

Разпечатка 7. Използване на оператор `||` („логическо или“)

Хипотетичната програма, от която даденият сегмент на разпечатката е извадка, определя вероятните високорискови COVID пациенти на база тяхната възраст и тегло. За да е високорисков, хипотетичният пациент трябва да отговаря на едно от двете условия – да е над 60 годишен или да тежи над 90 килограма.

Изчислението на оператора, демонстриран на ред 8, протича по подобен начин, както при „логическото и“. Първо се изчислява изразът отляво на оператора. Ако изразът е **true**, това е достатъчно, за да предопредели крайния

резултат, който в този случай ще бъде също **true**. Ако обаче левият израз е **false**, трябва да се изчисли и израза отдясно на оператора. Крайният резултат се получава както е указано в таблицата.

Операторът „логическо не“ е пример за унарен оператор, т.е. такъв който има само един операнд. „Логическото не“ се изписва с единична удивителна ! пред операнда, който може да е променлива или литерал от тип **boolean**.

„Логическото не“ обръща логическата стойност. Т.е. ако операндът е **true**, резултатът от изпълнение е **false** и обратното. Това е обобщено в долната таблица.

операнд	резултат
false	true
true	false

Пример за използването на „логическо не“ е даден в следващата разпечатка:

6	<code>int age = 16;</code>
7	<code>boolean minor = age<=18; // minor e true</code>
8	<code>boolean adult = !minor; // adult e false</code>
9	<code>boolean voter = !(age<18); // voter e false</code>

Разпечатка 8. Използване оператор ! („логическо не“)

В примера се определят три характеристики на даден човек. Променливата **minor** (непълнолетен) получава стойност **true** или **false** в зависимост от това дали възрастта на човека е над 18 години. Променливата **adult** (пълнолетен) получава **true** ако човекът е пълнолетен. Двете променливи задължително съдържат противоположни логически стойности и ние можем да се възползваме от това чрез оператора „логическо не“.

На ред 18 правим проверка на възрастта и попълваме стойността на *minor*. На ред 19 използваме логическото „не“ за да обърнем стойността на *minor* и да попълним с резултата променливата **adult**. По този начин спестяваме извършването на още една проверка, която би изглеждала по същия начин с „обърнат“ оператор за сравнение, и по този начин правим програмата по-ясна.

Ред 9 демонстрира, че логическото „не“ може да стои пред цели логически изрази.

Както бе споменато, операторите **&&** и **||** прекратяват изчислението на десния операнд, ако левият е достатъчен за определяне на резултата. Този начин на работа се нарича *мързеливо изчисление* или *мързелива евалуация*.

Мързеливата евалуация в повечето случаи не е проблем. Тя трябва да се познава обаче, тъй като при по-сложни изрази в тях могат да бъдат включени методи, които извършват определени действия, след което връщат стойност от тип **boolean**. Ако такъв метод се намира от дясната страна на **&&** и **||**, и при положение че лявата е достатъчна за определяне на крайния резултат, той няма да бъде извикан и действията няма да бъдат извършени.

Малко известен факт е това, че ако вместо **&&** и **||** се използват операторите **&** и **|**, ефектът ще бъде същия с тази разлика, че и двата операнда ще бъдат изчислени задължително, т.е. няма да имаме мързелива евалуация. Операторите **&** и **|** са познати като *побитово „и“* и *побитово „или“* и нормално се използват върху целочислени типове. Използвани върху логически стойности обаче, те действат като логически оператори.

Освен побитовото „и“ и побитовото „или“, има и побитов оператор „изключващо или“, който се отбелязва с **^**. По същия начин, когато се използва върху логически стойности, този оператор действа като логически оператор, който действа както е описано в следващата таблица:

операнд 1	операнд 2	резултат
false	false	false
true	false	true
false	true	true
true	true	false

Действието на побитовото „изключващо или“ като логически оператор наподобява „изключващото или“ с тази разлика, че когато и двата операнда са **true**, резултатът е **false**. С други думи този оператор настоява стриктно само едно от двете условия да бъде изпълнено, за да върне резултат **true**.

За определяне на резултата от „изключващо или“ трябва задължително да бъдат известни стойностите и на двата операнда, поради което той няма версия с мързеливо оценяване.

4. Побитови оператори

Побитовите оператори в Java се изпълняват основно върху променливи и литерали от целочислен тип. Тези оператори действат върху отделните битове на двоичното представяне на числата, откъдето вземат името си.

Операторът побитово „и“ се обозначава с единичен знак амперсанд (&). Действието на оператора за побитово „и“ може да се опише със следната таблица:

бит 1	бит 2	резултат
0	0	0
0	1	0
1	0	0
1	1	1

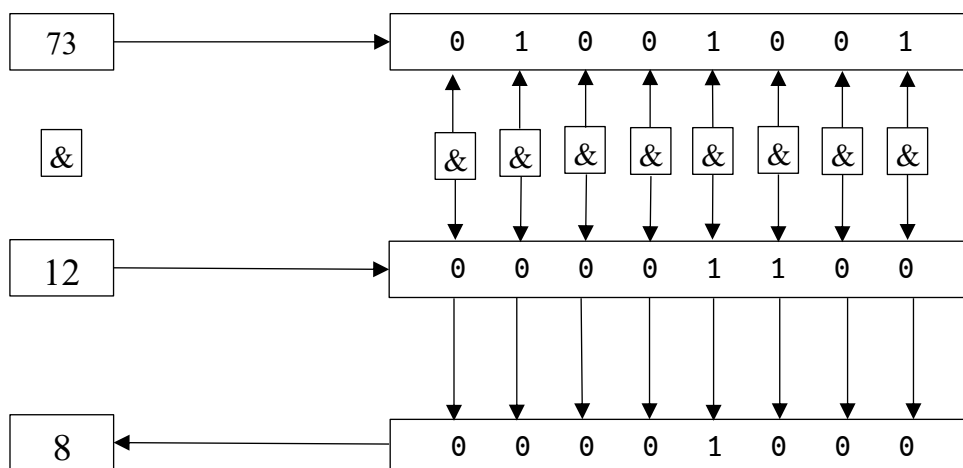
Ако читателят замести мислено 0 с false и 1 с true, той ще получи таблицата на оператора „логическо и“, разгледан по-горе. Долният пример демонстрира резултата от прилагане на „побитово и“ върху две цели числа:

```
1 package uk.co.lalev.inputoutput;
2
3 public class Main {
4     public static void main(String[] args) {
5         byte chunk=73;
6         byte box=12;
7         int result = chunk & box;
8         System.out.println(result); // 8
9     }
10 }
```

Разпечатка 9. Използване оператор & („побитово и“)

Програмата на разпечатката прилага „побитово и“ върху две числа от тип *byte* – 73 и 12 – и получава резултат 8.

Долната диаграма показва как е получен този резултат:



Фигура 1. Действие на оператора „побитово и“

Както е видно от диаграмата, двоичното представяне на 73 в компютъра е 01001001, докато 12 се представя като 00001100. Читателят може да приложи горната таблица върху отделните двойки битове, както са показани на фиг. 3.1 и да се убеди, че резултатът е 00001000, което е 8 в десетична бройна система.

Действието на „побитово или“, който както вече споменахме, се задава със знака единична вертикална черта |. Неговото действие е сходно с това на логическия му аналог, с тази разлика, че този оператор работи върху отделни двойки битове идващи от целочислените типове в Java.

бит 1	бит 2	резултат
0	0	0
0	1	1
1	0	1
1	1	1

Действието на „изключващото или“ (задавано с ^) като побитов оператор е описано в следващата таблица:

бит 1	бит 2	резултат
0	0	0
0	1	1
1	0	1
1	1	0

Последният побитов оператор е оператора „допълнение до две“. Този оператор също произтича от двоичната аритметика и се отбелязва с единичен знак тилда (~). Той е унарен, т.е. работи само върху един операнд. Без да влизаме в детайли, този оператор създава такова число, което събрано с оригиналния операнд произвежда препълване. Това препълване има такъв характер, че произвежда нули във всички битове на резултата.

Както допълнението до две, така и другите побитови оператори изглеждат доста странно на начинаещите програмисти. Те обаче имат своето приложение в множество утвърдени компютърни алгоритми от всички области на информатиката, включително криптографията, оптимизационните изчисления и пр. Свидетелство за тяхната универсалност е и това, че самите компютри и техните процесори са съставени от електронни блокове, които изпълняват точно същите операции върху електрически сигнали, представляващи двоични нули и единици.

5. Оператори за побитово изместване

Операторите за побитово изместване преместват битовете в числото наляво или надясно. Операторът за побитово изместване наляво се обозначава с <<. Операторът премества всички битове в двоичното представяне с една или повече позиции наляво. Празните позиции отдясно, които се образуват по този начин, се попълват с нули. Позициите „изпадащи“ от левия край на числото се отрязват.

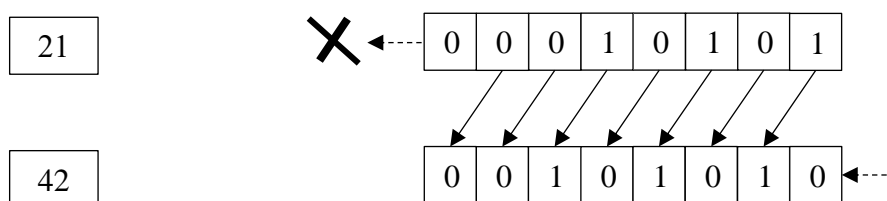
Долният пример демонстрира използването на оператора:

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.println(21<<1); // (Една позиция наляво) 42
6         System.out.println(3<<2);  // (Две позици наляво) 12
7         System.out.println(5<<3);  // (Три позиции наляво) 40
8         System.out.println(2147483647); // 0b01111111111111111111111111111111
9                                         // 2147483647
```

10	System.out.println(2147483647<<1); // 0b11111111111111111111111111111110
11	// -2
12	}
13	}

Разпечатка 10. Използване на оператори за побитово изместване

Фиг. 3. илюстрира действието на оператора на ред 5 на разпечатката. Вляво са показани десетчните стойности, докато вдясно са показани двоичните представяния на числата.



Фигура 3. Побитово изместване наляво с един бит върху стойност от тип *byte*.

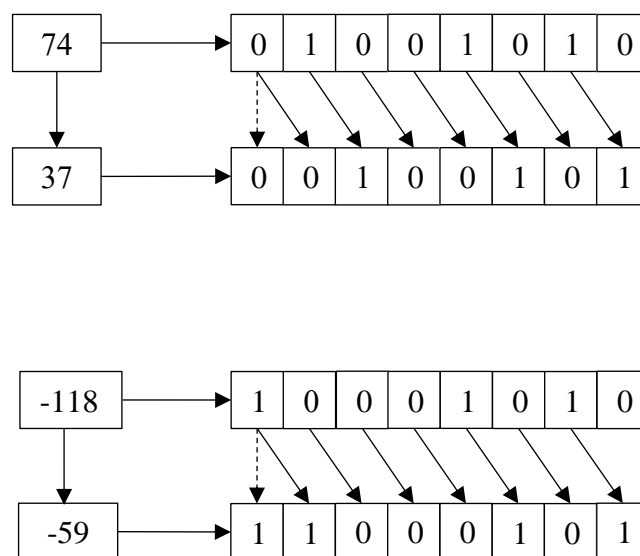
Както е видно от разпечатката и фигурата, двоичното изместване надясно е еквивалентно на умножение по две за всяка позиция, с която се измества резултатата надясно, но при положение че няма препълване.

Няма да дискутираме всички аспекти на препълванията, тъй като те са доста сложна материя с множество особености, която е тясно свързана с двоичното представяне на числата в компютъра. Така например фиг. 3.2 показва ситуацията, когато стойността е от тип **byte**. Литералът на ред 5 от разпечатката обаче е от тип **int** (както вече споменахме, всички немаркирани целочислени литерали в Java са от този тип). Въпреки това резултатите са едни и същи, тъй като числата 21 и 42 се поместват без проблеми както в **byte**, така и в **int** и всички изпадащи отляво цифри са двоични нули независимо кой от двата типа се използва.

Редове 8 и 10 демонстрират един на пръв поглед странен ефект от препълването, при което единица навлиза в бита за знак, който до момента е бил нула. На ред 8 е отпечатана максималната положителна стойност за литерал **int**.

Тя е 2147483647. В двоична бройна система това се изразява с една нула, следвана от 31 бита единици. При изместване надясно с една позиция, числото вече започва с двоична единица, което всъщност обозначава отрицателно число. Така изместената с едно наляво последователност от битове на ред 10 всъщност представлява -2 в компютъра.

Операторът за изместване надясно със запазване на знака се обозначава с `>>`. Той взема два операнда. Левият операнд е числото, върху чието двоично представяне ще се работи, докато десният операнд показва с колко бита надясно ще бъде изместено числото. При изместване надясно, най-малко значещите битове⁸ „изпадат“ или биват отрязани от числото. В областта на най-значещите битове се образуват празни позиции. За да се запази знака на числото, тези позиции се запълват с двоични нули или единици съответно за положителните или отрицателните числа (вж. фиг. 4).

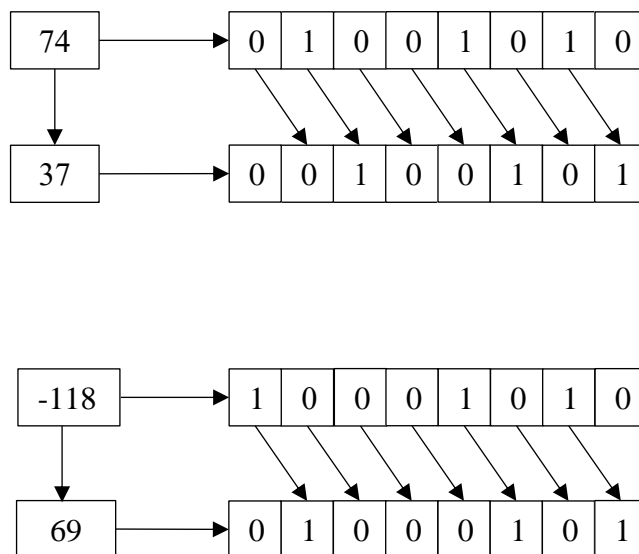


Фигура 4. Изместване надясно с единица при оператор `>>`.

Операторът за побитово изместване надясно без запазване на знака се обозначава с `>>>` в Java. Той действа по същи начин като предходния оператор, с

⁸ Най-малко значещите битове са еквивалентни на най-младшите разряди на десетичното число. В десетичното число най-малко значи цифрата на единиците, следвана от цифрата на десетиците и т.н.

изключение на това, че позициите, които се освобождават отдясно се попълват винаги с нули (вж. фиг. 5).



Фигура 5. Изместване надясно с единица с оператор `>>>`

6. Троичен оператор

Троичният оператор позволява изчислението на даден израз да бъде извършено по два алтернативни начина, в зависимост на настъпването на дадено условие. Синтаксисът на троичния оператор е:

условие или лог. стойност ? израз1 : израз2

Условието може да бъде израз с оператори за сравнение и логически оператори. Ако то се изчисли до логическата стойност **true**, виртуалната машина ще изчисли *израз1* и ще замени цялата конструкция с получения резултат. Ако условието се изчисли до логическата стойност **false**, виртуалната машина ще изчисли *израз2*.

Долният пример демонстрира използването на троичния оператор:

```

6  boolean male = false;
7  double weight = 64;
8  double AverageHeightMale = 180;
9  double AverageHeightFemale = 160;
10
11  double approxExcessWeight = weight -
12    (male ? AverageHeightMale - 100 : AverageHeightFemale - 100);
13
14  System.out.println(approxExcessWeight); // 4.0
15
16  boolean citizen = true;
17  double yearlyIncome = 50000;
18
19  double tax = citizen ? yearlyIncome*0.1 : 0;
20
21  System.out.println(tax); // 5000.0
22
23  double a = -2;
24  double b = -3;
25  double c = 2;
26  double D = Math.pow(b,2)-4*a*c;
27
28  System.out.print("Корените на квадр. у-нине са: " +
29    (D>=0 ? (-b + Math.sqrt(D)) / (2*a) +
30    " и " + (-b - Math.sqrt(D)) / (2*a) : "N/A") // -2.0 и 0.5
31  );

```

Разпечатка 11. Използване на троичен оператор

Редове 6-12 от програмния сегмент на разпечатка 11 демонстрират първата употреба на троичен оператор. В случая се изчислява наднорменото тегло на даден човек по много приблизителна формула, която определя нормалното тегло като изважда 100 от средния ръст на мъжете и жените. Тъй като изчислението е различно за мъжете и жените (техният среден ръст е различен), ние използваме троичния оператор за да изчислим едно от двете здравословни тегла. Те са съответно **AverageHeightMale - 100** за мъжете и **AverageHeightFemale – 100** за жените. Тъй като в нашия пример стойността на **male** е **false**, което индикира, че човекът не е мъж, троичният оператор ще изчисли втората формула и ще получи здравословното тегло за жените. Това в примера конкретно ще произведе 60. Това число ще продължи в изчислението на мястото на целия троичен оператор. Оставащата част от изчислението след изчисляването на троичния оператор ще изглежда като **weight-60** и очаквано ще произведе резултат 4, който ще бъде присвоен на променливата **approxExcessWeight**.

Вторият пример касае доста хипотетичния сценарий за данъчно облагане при което доходите на гражданите на държавата се облагат с 10% данък, докато доходите на чужденците не се облагат. На редове 16 и 17 се настройват променливите, които ще участват в примера. Променливата **citizen** получава стойност **true**, което отразява факта, че облаганият човек е гражданин. Променливата **income** получава неговия хипотетичен доход.

Същинският троичен оператор е на ред 19. Тъй като **citizen** е **true**, всъщност ще се изчисли само изразът **yearlyIncome*0.1**. С неговата стойност ще бъде заместен целият троичен оператор и изчислението ще продължи нататък. Тъй като няма други изчисления за извършване, стойността на изразът **yearlyIncome*0.1** (5000) ще бъде присвоена на променливата **tax**.

Последният пример в сегмента демонстрира решаване на квадратно уравнение с коефициенти **a**, **b** и **c**. Той демонстрира, че когато логиката на изчисления стане по-сложна, троичният оператор може да стане сравнително сложен за четене. Ето защо начинаещите програмисти не трябва да прекаляват с употребата му, тъй като в повечето случаи изчисляваната формула може да бъде разделена на части, които да се изчислят избираемо с помощта на конструкцията **if**, която ще бъде разгледана в следващата тема. Така обикновено се получава много по четлив код.

Разчитането на примера трябва да започне от ред 20, където се изчислява дискриминантата на уравнението. Ако тази стойност е отрицателна, уравнението няма реални корени. Ето защо троичният оператор на ред 23 проверява именно това. Ако проверката **D>=0** върне **false**, ще бъде изчислен втория израз, който е просто символния низ **N/A**. Той ще бъде долепен към фразата на ред 28 за да обозначи, че уравнението няма реални корени. Ако **D>=0** се изчисли до **true**, това означава, че дискриминантата е неотрицателна и уравнението има един или два различни реални корена. В такъв случай троичният оператор ще изчисли изразът

$$(-b + \text{Math.sqrt}(D)) / (2*a) + " \text{ и } " + (-b - \text{Math.sqrt}(D)) / (2*a)$$

Двата компонента, разположени от двете страни на символния низ „и“, изчисляват двата корена за да формират изведения на конзолата низ, който е показан в коментара на ред 30.

7. Оператори за увеличаване / намаляване с единица

В Java има четири унарни оператора, които съкращават писането на програмен код, когато става дума за увеличаване или намаляване на променлива с единица.

Операторът за **преинкремент** се отбелязва с `++` преди името на променлива в израз (например `++a`, `++value` и т.н.) и предизвиква увеличаването на променливата с единица. Увеличената стойност взема участие в по-нататъшното изчисление на израза.

Операторът за **постинкремент** се отбелязва с `++` след името на променливата в израз (например `a++`, `value++` и т.н.). Операторът предизвиква увеличаването на променливата с единица. В по-нататъшното изчисляване на израза обаче взема оригиналната стойност на променливата.

Операторът за **предекремент** се отбелязва с `--` преди името на променлива в израз (например `--a`, `--value` и т.н.). Операторът предизвиква намаляването на променливата с единица. В по-нататъшното изчисляване на израза обаче взема оригиналната стойност на променливата.

Операторът за **постдекремент** се отбелязва с `--` след името на променливата в израз (например `a--`, `value--` и т.н.). Операторът предизвиква намаляването на променливата с единица. В по-нататъшното изчисляване на израза обаче взема оригиналната стойност на променливата.

Долният пример демонстрира ефектите от прилагането на четирите оператора:

6	<code>int a = 5; // a->5</code>
7	<code>int b = a++; // a->6, b->5</code>
8	<code>int c = b--; // a->6, b->4, c->5</code>

9	<code>int d = --a; // a->5, b->4, c->5, d->5</code>
10	<code>int e = ++d+1; // a->5, b->4, c->5, d->6, e->7</code>
11	<code>b = --a++; // Грешка при компилация</code>

Разпечатка 12. Оператори за увеличаване/намаляване с единица

На ред 6 от програмния сегмент в горния пример, стойността на променливата **a** се установява на 5. Първият оператор за увеличаване е постинкремент операторът на ред 7. Благодарение на него, променливата **a** се увеличава с едно до 6, а старата ѝ стойност участва в израза от дясната страна на оператора за присвояване. Този израз няма други елементи, поради което стойността на **b** става 5.

Постдекремент операторът на ред 8 намаля стойността на **b**, но не преди тази стойност да бъде взета за изчисляване на израза отдясно на оператора за присвояване. По този начин стойността на **c** става 5. По-интересен е преинкремент операторът на ред 10, приложен към променливата **d**. Тази променлива преди изпълнението на ред 10 има стойност 5. Преинкремент операторът първо я увеличава до 6 и тази стойност участва в изчислението на израза по-нататък. За разлика от изразите на предните редове, този път изразът има още елементи – аритметичен оператор за събиране. Евентуално този израз се изчислява до 7, което става стойност на променливата **e**.

Ред 11 демонстрира, че не е възможно да имаме едновременно пре- и пост-оператор на даден операнд. Това предизвиква грешка при компилация.

8. Комбинирани оператори за присвояване

Java предлага още механизми за съкращаване на писането на изрази, когато те засягат състоянието на само една променлива и се състоят само от един оператор. Такива изрази например могат да бъдат **value=value+5**, **a=a>>3** и т.н.

Механизмите идват под формата на комбинираните оператори за присвояване, показани в следващата таблица:

оператор	пример	действие
+= -= *= /=	a+=5 a-=3 value*=2 test/=3.5	Добавя, изважда, умножава или дели стойността на указаната променлива (според знака в оператора) и присвоява резултата на същата променлива.
%=	a%=3	Изчислява остатък при деление на стойността на променливата и остатъкът става нова стойност на тази променлива.
>>= >>>= <<= = &= ^= ~=	value>>=3 a>>>=3 b<<=1 a&=0b0001 value&=15	Извършва указаната побитова (или логическа) операция върху стойността на променливата и присвоява резултата на същата променлива.

9. Приоритет на операторите в Java

На практика всеки път, когато в точката до момента давахме примери за изрази, съдържащи повече от един оператор, ние се озовавахме пред въпроса какъв точно е редът на изпълнение на операторите. Java (както повечето програмни езици) се придържа към приоритета на изпълнение на операторите, посочен в долната таблица.

променлива++ променлива--
++променлива --променлива +израз -израз ! ~ оператор за превръщане на типове
* / %
+ -
<< >> >>>
> < <= >= instanceof
== !=
&
^

&&
= += -= *= /= %= &= ^= = <<= >>= >>>=

При наличие на множество оператори в един израз, Java изпълнява първо операторите, посочени в първия ред на таблицата, след това във втория ред и т.н. Ако в даден момент Java трябва да избере между два или повече оператора от един и същи приоритет, Java ги изпълнява отляво-надясно. Фразата „отляво-надясно“ е само приблизително описание на точния процес, който може да се опише по-прецизно по следния начин:

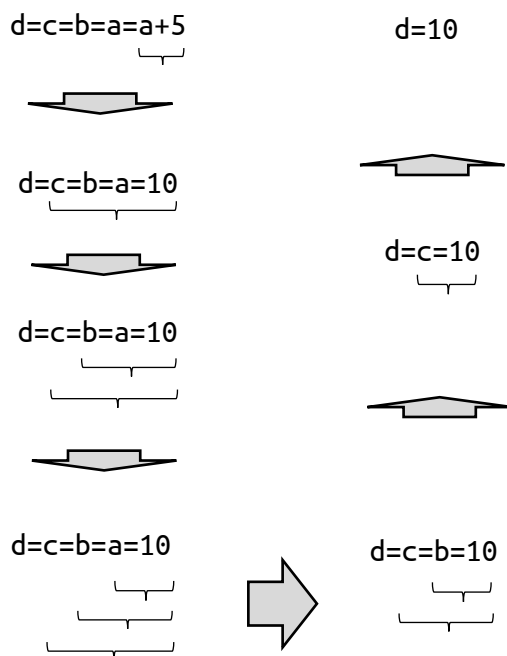
При наличие на израз с множество оператори, той се разделя на две части при оператора с *най-нисък* приоритет“. Двете части се изчисляват рекурсивно като отделни изрази, които условно наричаме „лява“ и „дясна“ страна. При определяне на стойността на лявата и дясната страна на оператора, се изчислява резултата от неговото изпълнение. Някои логически оператори не изчисляват винаги дясната страна, както бе пояснено горе. Унарните оператори нямат лява страна. Операторите за присвояване не изчисляват лява страна, тъй като тя е променлива, която ще съхрани стойността на дясната страна.

Този рекурсивен елемент при изпълнението отляво-надясно е илюстриран със следния пример:

6	int a = 5;
7	int b, c, d, e;
8	d = c = b = a + 5;

Разпечатка 13. Изпълнение на оператори с еднакъв приоритет

Процесът на изчисляване на израза на ред 8 минава през няколко стъпки. Първо се изчислява стойността на `a+5`. Резултатът е 10. Временното състояние на израза към този момент на изчислението изглежда е `d=c=b=a=10` (вж. фиг. 3.1).



Фигура 6. Изчисление на израза от ред 10 на разпечатка 3.13

Тъй като всички оставащи оператори към този момент са оператори за присвояване, Java ще ги изпълни отляво-надясно. За да присвои стойност на `d`, Java ще изчисли първо дясната страна, т.е. `c=b=a=10`. Това на свой ред е сложен израз с множество оператори. Най-левият оператор за присвояване ще бъде изпълнен първи, поради което Java ще изчисли неговата дясна страна - `b=a=10`. Дясната страна на този израз е `a=10`.

Така в процеса на изчисление най-накрая достигаме до израз, който съдържа един оператор и може да бъде изчислен веднага. Изчислението на `a=10` ще доведе до присвояване на стойност 10 на променливата `a`. `a=10` ще бъде заместена с върнатата от оператора стойност, а именно – 10 – дясната страна на оператора за присвояване.

Към този момент изразът $b=a=10$ вече е трансформиран до $b=10$ и може да бъде изчислен веднага. Резултатът е присвояване на стойност 10 на променливата b и заместване на израза с 10, което трансформира $c=b=a=10$ в $c=10$. Така накрая d получава също стойност 10.

При сложни изрази е полезно да се използват скоби. Скобите указват различен приоритет на извършването на действията, също както правят това в математическите изрази. При наличие на скоби, изпълнението ще започне от частта на израза, която е в най-вътрешните скоби.

Върху горната таблица могат да се направят още коментари. Така например в таблицата е посочен операторът *instanceof*, който не е дискутиран до момента. Този оператор определя дали дадена променлива съдържа референция към обект от дадения тип или негов наследник. Този оператор е свързан с обектно-ориентираното програмиране, поради което ще бъде разгледан като в следващите теми бъдат разгледани обектите и класовете.

Друго важно наблюдение е това, че аритметичните оператори $+$ и $-$ имат унарна и бинарна версия. В израза $a = -b + 4$ минусът пред b е унарен оператор, докато плюсьт между b и 4 е бинарен. Унарните оператори имат по-висок приоритет от бинарните, както е показано по-горе. Поради това при изчисление на горния израз първо ще бъде взета стойността на b с обратен знак и след това към нея ще бъде добавена стойността 4.

10. Превръщания между стойности в контекста на изчисляване на изрази

В предната тема ние разгледахме в основни линии правилата, които управляват превръщанията между елементарните типове, когато се използва оператор за присвояване. Изчисляването на изрази, които включват други оператори също подлежи на определени правила, имащи за цел уеднаквяването

на типовете, с които се пресмята. Тези правила са само две и засягат само числовите типове:

1. Когато един от двата операнда на аритметични унарни или бинарни оператори, както и на оператори за побитова манипулация и изместване, е от тип **byte**, **char** или **short**, той се превръща в операнд от тип **int**. **int** има достатъчно битовете, за да помести всички възможни числа, които могат да се съхраняват в един от трите типа, така че това превръщане винаги има *разширяващ* характер и не може чрез прилагането му да се

2. Ако един от двата операнда на бинарен аритметичен оператор е **double**, другият операнд също се превръща в **double**. В противен случай, ако един от двата операнда е от тип **float**, то и другият операнд се превръща във **float**. Тези превръщания също имат разширяващ характер.

Правилата са до голяма степен интуитивни, но все пак водят до някои особени резултати, част от които са илюстрирани в програмния сегмент от следващата разпечатка:

```
6  int a = 7;
7  double b = a/2;
8  System.out.println(b); //!!! 3.0 вместо 3.5
9  b = a / (double) b;    // правилно
10 b = (double) a / b;    // също правилно
11
12 byte c = 5;
13 byte d = c;
14 byte e = c+d;          // Грешка при компилация!
15 e = (byte)c + d;       // Грешка при компилация!
16 e = c + (byte)d;       // Грешка при компилация!
17 e = (byte)(c+d);       // правилно
```

Разпечатка 14. Превъщане на типове в контекста на изчисляване на изрази

На редове 6-8 на променливата **a** се присвоява стойност 7, а на променливата **b** се присвоява стойност **a/2**. Резултатът би трябвало да бъде дробно число – **3.5**. Вместо това, както ще бъде изведено на ред 8, резултатът ще бъде **3.0**. Причината не е в типа на **b**. **b** е от тип **double** и със сигурност може да съхранява дробни числа. Вместо това причината е в това, че и двата операнда на делението на ред 10 са **int** и Java няма да ги превърне в **double** автоматично. Това

поведение не е резултат на недомислица и позволява лесно да се изпълнява целочислено деление, което е важен компонент от множество компютърни алгоритми. При деление на 7 на 2, целочисленото деление дава 3. Този резултат се превръща в **double** с разширяващо превръщане, дискутирано в предната тема. Това произвежда крайният резултат – **3.0**.

Програмист, който иска да раздели две цели числа и да получи резултат от тип **double** или **float**, трябва да превърне поне един от давата операнда съответно във **float** или **double** чрез експлицитен каст, както е показано на ред 9. Тогава, съгласно очертаните по-горе правила, Java ще превърне и другия операнд. Това ще доведе до това, че крайният резултат също ще бъде от желанния тип – **float** или **double**.

Редове 12-14 демонстрират неинтуитивните ефекти от това, че всички литерали и променливи от тип **byte**, **char** и **short** се превръщат в **int**, когато участват в изрази с аритметични или побитови оператори. Съгласно това правило, и двете променливи на ред 14 се превръщат в **int**. Това не променя стойността им, но резултатът от събирането сега е от тип **int** и не може да се помести в променливата **e**, която е от тип **byte**.

Ако държим да поставим резултата в променлива от такъв тип, трябва да направим както е показано на ред 17. Да превърнем типа на резултата от **int** към **byte** преди да го присвоим на променливата. Това е стесняващо превръщане и трябва да се внимава да не се получи отрязване на резултата.

Редове 15 и 16 показват два грешни начина да се направи превръщането. На тези редове операторът за каст е приложен само към единия от операндите. Това не променя типа му, тъй като и двата са от тип **byte**, а оператора за каст се изпълнява преди аритметичния оператор, който ще направи превръщането към **int**. Крайният резултат на редове 15 и 16 е отново стойност от тип **int**, която не може да се помести в **byte**.

Глава 4. Условни конструкции и цикли

Изпълнението на типичната компютърна програма рядко протича праволинейно. Всички достатъчно сложни програми имат своеобразни „разклонения“, които се изпълняват само когато определени условия са изпълнени. Програмната логика на повечето програми също така често налага връщане назад и многократното изпълняване на определени сегменти.

Като груб пример за необходимостта от разклоняване на изпълнението на програмата може да се даде изчисляването на лихви върху дадена сума, което типично става само когато плащането на сумата се забави във времето. По подобен начин проверката за валидно въведено ЕГН типично се прави само ако въведеното лице е български гражданин и има ЕГН.

Повторенията на дадени сегменти от програмата се правят най-често когато трябва да се обработи по един и същи начин информацията на множество еднакви обекти, например служители в дадената компания.

Декларациите за разклоняване на протичането на програмата на професионални език често се наричат „**условни конструкции**“ или „**декларации за контрол на протичането на програмата**“ и в Java те са представени от конструктите **if-else** и **switch**.

Декларациите за повторение на определен сегмент от програмата се наричат „**декларации за реализиране на цикли**“, тъй като професионалното понятие за подобно повторение е именно „**цикъл**“. Тези декларации в Java са представени от две форми на конструкцията **for** и две форми на конструкцията **while**, които са допълнени от две специални ключови думи - **break** и **continue**.

1. Конструкция if-else

Основната конструкция, която се използва за разклоняване на протичането на Java програмите е декларацията **if-else**. Тя има следния общ вид:

```
if (логически израз) {  
    ...  
    // декларации, които се изпълняват, ако изразът е true  
    ...  
} else {  
    ...  
    // декларации, които се изпълняват, ако изразът е false  
    ...  
}
```

Използването на **if-else** позволява разделянето на програмата на две алтернативни разклонения. Първото разклонение се задава в първите фигурни скоби (първият блок от код) и се изпълнява само ако условието, зададено с логическия израз, е изпълнено. В случай че условието не е изпълнено, се изпълнява само вторият клон, който е зададен във вторите фигурни скоби след **else**.

Пример за много просто разклонение е изчисляването на абсолютна стойност, показано в следващия пример:

```
1 import java.util.Scanner;  
2  
3 public class Main {  
4  
5     public static void main(String[] args) {  
6         System.out.println("Въведете число: ");  
7         int value = new Scanner(System.in).nextInt();  
8  
9         if (value >= 0) {  
10            System.out.print("Абсолютната стойност на "+value+" е: ");  
11            System.out.println(value);  
12        } else {  
13            System.out.print("Абсолютната стойност на "+value+" е: ");  
14            System.out.println(-value);  
15        }  
16    }  
17 }
```

Разпечатка 1. Програма с разклонение

Програмата на разпечатка 1 има непознати елементи, които касаят вход и изход от конзолата. На редове 6 и 7 потребителят въвежда число, което се поставя в променливата **value**. Конструкцията **if-else** е поместена на редове 9 до 15 и се интерпретира по следния начин от Java:

Ако стойността на **value** е по-голяма от 0, се изпълняват редове 10 и 11. Ако е вярно обратното (т.е. **value** не е по-голямо от 0) се изпълняват редове 13 и 14. Като резултат от изпълнението на едно от двете разклонения, потребителят винаги получава абсолютната стойност на въведеното число. Така например ако той въведе 14, **value>=0** е **true** и изпълнението на редове 10 и 11 води до извеждането на текста *“Абсолютната стойност на 14 е: 14”*. Ако потребителят въведе -14, тогава **value>=0** е **false** и се изпълняват редове 13 и 14. Те извеждат *“Абсолютната стойност на -14 е: 14”*.

Логическият израз, който при **if** винаги се записва в скоби, винаги връща **true** или **false** и типично съдържа някои от разгледаните в предната глава оператори за сравнение. В примера на разпечатка 1 това е операторът „по-голямо или равно“, но на практика са допустими всякакви логически изрази, включително фиксирани логически константи (**true** или **false**), функции или променливи от тип **boolean** и всякаква комбинация от тях, включваща оператори за сравнение и логически оператори.

Използването на логически оператори е особено важно, тъй като позволява съставянето на по-сложни проверки, на чиято база да се определи кое разклонение на програмата да бъде използвано. Ето защо то е специално демонстрирано в следващия пример:

```
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Въведете възраст: ");
7         int age = new Scanner(System.in).nextInt();
8
9         if (age>=6 && age<=18) {
10             System.out.println("Вие сте ученик!");
11         } else {
12             System.out.println("Вие най-вероятно не сте ученик!");
13         }
14     }
15 }
```

Разпечатка 2. Програма с разклонение

Читателите могат да проверят, че логическият израз в скобите на **if** конструкцията е **true** само когато и двете условия, зададени с операторите за сравнение, са изпълнени. Така когато потребителят въведе възраст между 6 и 18 години, ще бъде изпълнен ред 10. В противен случай ще бъде изпълнен ред 12.

Не е задължително всяка **if** конструкция да има **else** част. Много често в програмите дадено действие се извършва при наличието на дадено условие, но когато условието не е вярно, не се предвиждат никакви действия. Този съкратен вид на *if-else* конструкцията има следния общ вид:

```
if (логически израз) {  
    ...  
    // декларации, които се изпълняват, ако изразът е true  
    ...  
}
```

Използването на **if** по този начин е демонстрирано на следващата разпечатка:

```
1 import java.util.Scanner;  
2  
3 public class Main {  
4  
5     public static void main(String[] args) {  
6         System.out.print("Въведете размер на глобата: ");  
7  
8         double fine = new Scanner(System.in).nextDouble();  
9         /* Изпускането на else частта е напълно допустимо */  
10        // Задава таван на глобата  
11        if (fine>500) {  
12            fine = 500;  
13        }  
14        System.out.println("Размерът на глобата е: "+fine);  
15    }  
16 }
```

Разпечатка 3. **if** без **else** част

Програмата на разпечатка 3 кани потребителя да въведе размер на хипотетична глоба и използва **if** за да наложи таван на тази глоба. Ако потребителят въведе сума над 500, тя ще бъде отрязана до този таван. Тъй като отрязването на сумата на глобата се налага само когато тя е над тавана, ситуацията е подходяща за изпускане на **else** частта. Резултатната конструкция е поместена на редове 11-13.

Докато изпускането на **else** частта е напълно допустимо, първият блок от код, който се изпълнява когато логическият израз върне **true**, не може да бъде изпуснат. Това е демонстрирано в следващия пример:

```
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.print("Въведете размер на заплатата: ");
7
8         double salary = new Scanner(System.in).nextDouble();
9
10        /*
11         * Грешка при компилация, тъй като е изпуснат блока от
12         * код, който ще се изпълни в случай че salary>800 е true
13         */
14        // if (salary>800)
15        // else {
16        //     salary=800;
17        // }
18
19        /*
20         * Правилен, но не особено ефективен начин за реализация
21         * на минимум на работната заплата
22         */
23        if (salary>800) {
24            //празен блок от код
25        } else {
26            salary=800;
27        }
28
29        // По-добър начин
30        if (!(salary>800)) {
31            salary=800;
32        }
33
34        // Още по-добър начин
35        if (salary<800) {
36            salary=800;
37        }
38    }
```

Разпечатка 4. Демонстрация на **if** без **else** част

Програмата на разпечатка 4 е подобна на предната, с тази разлика, че се опитва да наложи минимум на работната заплата, въведена от потребителя. Ако въведената стойност е под 800 програмата просто поставя за нова стойност 800.

Първият начин за реализация (редове 14-17) изпуска напълно първия блок и преминава директно към **else** частта. Това не е позволено в Java и кодът е коментиран, тъй като ще доведе до грешка при компилация.

На редове 23-27 е демонстриран един вариант на корекция, който ще се компилира – а именно да се остави празен първият блок с код, който ще се изпълни ако условието е истина. Това обаче е излишно и затормозява писането на програмата. В такъв случай е много по-добре смисълът на логическото условие да бъде обърнат. Вместо програмистът да възлага проверката **salary>800**, той може да обърне смисъла на проверката като постави пред нея логическия оператор **!** (логическо “не”). Логическото “не” превръща **true** във **false** и **false** в **true** като по този начин позмолжава **else** частта да бъде разместена с първия блок в даден **if** оператор. Резултатът е демонстриран на редове 30-32. Докато логическото „не“ е лесен начин за „обръщане на смисъла“ на всякакви, потенциално сложни проверки, в дадения случай читателят лено може да види, че проверката с логическо „не“ на ред 30 е еквивалентна на проверката **salary<800**, което е най-чистият начин за изразяване на програмната логика, касаеща задаване на минимална стойност на въвежданата заплата.

Завършваме дискусията на разпечатка 4 с предупреждението, че когато се борави с логически проверки винаги трябва да се обръща внимание на „граничните“ случаи. В примера на разпечатка 4 **if** конструкцията на редове 35 до 37 не е напълно еквивалентна на конструкцията на редове 30-32. Това е така, тъй като конструкцията на редове 30-32 ще изпълни блока си от код при стойност на **salary** от точно 800. В същия случай конструкцията на редове 35-37 няма да изпълни блока си от код. За щастие в контекста на примера това не нарушава програмната логика – въведената стойност 800 остава непроменена след приключване на изпълнението и на двете **if...else** конструкции.

Наблюдателните читатели вече са забелязали, че конструкцията **if-else** се отграничава от съседните декларации по различен начин. След нея и след блоковете от код, включени в нея, не се изписват точки и запетаи. Това е така, тъй

като по принцип точката и запетаята след блоковете от код са опционални. Освен това компилаторът на Java може да разпознае **if-else** конструкцията като съставна декларация и да определи лесно нейното начало и нейния край.

Когато дадено разклонение на програмата се състои само от една декларация, не е нужно тя да се изписва във фигурни скоби при **if-else** конструкцията. Това поражда различно изглеждащи **if-else** конструкции, които са демонстрирани на следната разпечатка.

```
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         double fine = new Scanner(System.in).nextDouble();
7         if (fine>500) fine = 500;
8         int age = new Scanner(System.in).nextInt();
9         if ((age>=6) && (age<=18))
10            System.out.println("Вие сте ученик!");
11        else
12            System.out.println("Вие най-вероятно не сте ученик!");
13    }
14 }
```

Разпечатка 5. **if-else** без блокове от код

Читателите ще разпознаят **if-else** конструкциите, демонстрирани в предните разпечатки. Тук обаче сме се възползвали от възможността за съкращаване на писането на програмен код, предлагана от Java. Тъй като действието, което се предвижда ако **fine>500** е **true**, се описва само с една декларация (а именно **fine=500**), тук тази декларация не е заградена от фигурни скоби. В този случай точката и запетаята в края на реда са задължителни. Те не са част от **if-else** конструкцията, а отбелязват края на декларацията, която се изпълнява под условие. Всъщност нищо не пречи **fine=500** да се изпише на следващия ред и по този начин **if-else** конструкцията да се раздели на два реда. В този случай точката и запетаята трябва да бъдат поставени в края само на този следващ ред и не след условието, зададено в скобите на **if-else** конструкцията на по-горния.

Когато **if** операторът има **else** част, същите вече очертани правила важат и при нея. Ако **else** частта съдържа само една декларация, то тази декларация не трябва да се огранчава във фигурни скоби (вж. редове 10-13 на разпечатката). Точката и запетаята са задължителни както на ред 11, така и на ред 13.

Изпускането на фигурните скоби от **if-else** конструкцията създава опасност от въвеждането на синтактични грешки, които произвеждат валиден Java код и поради тази причина лесно могат да бъдат изпуснати от компилатора. Ето защо много начинаещи и напреднали програмисти предпочитат да изписват със фигурни скоби дори **if-else** конструкции, които съдържат една декларация във всяко разклонение.

Много често програмната логика *не може* да бъде описана само с две различни разклонения. За да се адресират ситуации, в които има повече от две алтернативни действия, често **if-else** конструкциите се влагат една в друга. Долният леко екстравагантен пример демонстрира подобна ситуация:

```
1  import java.util.Scanner;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.print("Въведете оценка: ");
7          double mark = new Scanner(System.in).nextDouble();
8
9          if (mark>5.50) {
10             System.out.println(mark+" -> A");
11          } else {
12              if (mark>=4.50) {
13                  System.out.println(mark+" -> B");
14              } else {
15                  if (mark>=3.50) {
16                      System.out.println(mark+" -> C");
17                  } else {
18                      if (mark>=3.25) {
19                          System.out.println(mark+" -> D");
20                      } else {
21                          if (mark>=3.00) {
22                              System.out.println(mark+" -> E");
23                          } else {
24                              System.out.println(mark+" -> F");
25                          }
26                      }
27                  }
28              }
29          }
30      }
```


31	}
----	---

Разпечатка 6. Превръщане на оценка в ECTS

Целта на програмата от разпечатка 6 е да превърне въведена от потребителя числова оценка в буква, предвиждана от Европейската система за трансфер на кредити. Превръщането следва да се извърши както е описано в следващата таблица:

от 5.50 включително до 6	A
от 4.50 включително до 5.50	B
от 3.50 включително до 4.50	C
от 3.25 включително до 3.50	D
от 3.00 включително до 3.25	E
под 3.00	F

Стратегията, избрана от програмиста е последователно елиминиране на възможните случаи. С първата проверка на ред 9, която е част от първата **if-else** конструкция, програмистът проверява дали е настъпила първата ситуация – т.е. оценката е над 5.50, което означава, че буквата по ECTS ще бъде A. Ако проверката на ред 9 е **true**, ще се изпълни само блокът от код на редове 9-11 и целият блок от код към **else** частта на първата **if-else** конструкция (редове 11-29) ще бъде изпуснат без да бъде изпълнен.

Ако се окаже обаче, че стойността на **mark** е под 5.00, ще се изпълни само **else** частта (редове 11-29). Тази **else** част съдържа в блока си от код само една **if-else** конструкция, която на свой ред има свои части. Ние често наричаме подобно вмъкване „вложен **if**“. С втория **if**, програмистът проверява за втория случай, а

именно за оценка между 4.50 и 5.50. Проверката във вложения **if** (ред 12) проверява само дали **mark** ≥ 4.50 . Проверката **mark** < 5.50 не е необходима. Ако **mark** не беше под 5.50, **else** частта на първия **if** нямаше да се изпълни и така нямаше да достигнем до кода и проверката на ред 12. Тоест щом сме стигнали до изпълнението на ред 12, това значи че стойността на **mark** е под 5.50.

Ако тази втора проверка даде резултат **true**, програмата ще изведе ECTS оценка B. Ако обаче и втората проверка даде резултат **false**, е ясно, че въведената стойност е под 4.50. Ето защо във **else** частта на този **if** има вложен още един, трети **if**, който проверява за следващия случай. Това продължава до най-вътрешния **if** на редове 21-25. Проверката **mark** ≥ 3.00 на ред 21 проверява за наличието на предпоследния случай, при който ECTS оценката е E. Ако и тази проверка излезе **false**, всички варианти освен последния са елиминирани. Единствената възможност в този случай е **mark** да има стойност под 3. Ето защо в **else** частта на последния вложен оператор не се прави проверка дали стойността на **mark** е по-малка от 3.00, а направо се извежда резултата, който съответства на последния вариант.

Тъй като в **else** частта на всяка **if-else** конструкция (освен последната такава на редове 21-25) има една-единствена декларация, а именно вложена **if-else** конструкция, някои от фигурните скоби могат да бъдат изпуснати. Това е единствената ситуация в която изпускането на тези скоби всъщност води до-по четлив и лесен за поддържане код. Долният пример демонстрира това:

```
1  import java.util.Scanner;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.print("Въведете оценка: ");
7          double mark = new Scanner(System.in).nextDouble();
8
9          if (mark > 5.50) {
10             System.out.println(mark + " -> A");
11         } else if (mark >= 4.50) {
12             System.out.println(mark + " -> B");
13         } else if (mark >= 3.50) {
14             System.out.println(mark + " -> C");
15         } else if (mark >= 3.25) {
16             System.out.println(mark + " -> D");
```

```

17         } else if (mark >= 3.00) {
18             System.out.println(mark + " -> E");
19         } else {
20             System.out.println(mark + " -> F");
21         }
22     }
23 }

```

Разпечатка 7. Преработка на програмата от разп. 6. и оптимизация на записването на вложените **if-else** конструкции

Програмата на разпечатка 7 е същата като тази на разпечатка 6 с тази разлика, че **else** частите *не* са оградени с фигурни скоби. Изпълнението на програмната логика още протича по съвсем същия начин. Ако първата проверка на ред 9 даде **false**, се изпълнява **else** частта на първия **if**, която се състои само от един вложен **if**, имащ своя **else** част с вложени по-нататък **if-else** конструкции. **else** частта на първия **if** се простира от ред 11 до ред 21. **else** частта на втория, вложен **if** по същия начин обхваща само една, на свой ред вложена **if-else** конструкция, която се простира на редове 13-21 и т.н.

Завършваме дискусията на **if-else** с важно наблюдение, касаещо инициализирането и декларирането на променливи в блоковете на конструкцията.

Ако дадена променлива бъде декларирана в някои от двата блока на **if-else** конструкцията, променливата ще е достъпна само в рамките на този блок. Освен това компилаторът на Java ще следи за променливи, които са дефинирани извън **if-else** конструкцията, но се инициализират с начална стойност в нея. Ако съществува път на изпълнение през **if-else** конструкцията, който не инициализира променливата, която след това се използва, компилаторът ще произведе съобщение за грешка. Двете ситуации са илюстрирани в долния пример:

```

1 public class Main {
2
3     public static void main(String[] args) {
4
5         boolean control = true;
6         int outerVariable;
7
8         if (control) {
9             outerVariable = 5;
10        } else {

```

```

11     int innerVariable = 5;
12 }
13
14 System.out.println(innerVariable); // Грешка при компилация!!!
15                                     // Променливата innerVariable на
16                                     // вече е недостъпна на този ред
17                                     // от програмата.
18 System.out.println(outerVariable); // Грешка при компилация!!!
19                                     // Променливата outerValue може
20                                     // да не е инициализирана.
21 }
22 }

```

Разпечатка 8. Деклариране и инициализиране на променливи в **if-else** конструкции

Примерната програма от разпечатка 8 декларира няколко променливи. Променливата **control** симулира логическа проверка. В реални програми тази проверка би била оформена като логически израз или функция, връщаща стойности от тип **boolean**, но тук просто получава стойност **true** на ред 5.

Втората променлива е **outerVariable**, която е дефинирана извън **if-else** конструкцията без да се задава начална стойност. Началната стойност се задава едва на ред 11, който се намира в блока от код, който отговаря на стойност **true** за променливата **control**. Тъй като **control** е **true** на пръв поглед изглежда, че блокът от код на редове 8-10 винаги ще се изпълни, докато блокът на **else** частта никога няма да се изпълни. Така променливата **outerVariable** винаги ще получи начална стойност. Компиляторът на Java обаче не влиза в предположения за възможните стойности на променливата **control**.

Компиляторът само забелязва, че ако стойността на **control** се окаже **false** при достигане на ред 8, ще се изпълни само **else** частта, която няма програмен код за инициализиране на променливата **outerVariable**. Това дава път на изпълнение на програмата, при която програмата достига ред 18 без **outerVariable** да е получила стойност.

На ред 18 програмистът прави опит да използва стойността на **outerVariable**, което почти гарантирано ще доведе до логическа грешка в програмата ако променливата не е получила начална стойност по-горе. За да

предпази програмиста от подобни грешки, Java, за разлика от други по-малко стриктни езици, ще откаже да компилира такъв код.

За да се преодолее грешката при компилиране програмистът трябва да постави в **else** частта код, който инициализира променливата. По този начин всички възможни разклонения на програмата, водещи до ред 18 ще инициализират на **outerVariable** със стойност.

Третата променлива в примера е декларирана на ред 11 в блока на **else** частта. Блоковете от код в **if-else** частта се третират като всички останали блокове с код. Като резултат дефинираните променливи в някои от тези блокове могат да се ползват само в рамките на блока. Ето защо опит за достъп до стойността на променливата **innerVariable** на ред 14 в примера също предизвиква грешка при компилация. Към този момент на изпълнение на програмата променливата **innerVariable** вече не съществува и виртуалната машина е освободила мястото в стека на програмата, заделено за съхраняване на нейната стойност.

Ако програмистът желае да използва стойност, получена от изчисление, което се намира в блок от код на **if-else** конструкция, и извън конструкцията, той трябва да дефинира променлива преди конструкцията и да присвои стойност на тази променлива вътре в конструкцията. В такъв случай, както бе току-що обяснено, програмистът трябва да внимава да не оставя възможен път през **if-else** конструкцията, който пропуска да зададе стойност на променливата.

2. Конструкция switch

Много често разклоненията в програмата, които трябва да бъдат направени на даден етап, са повече от две. Подобна ситуация може да бъде адресирана както бе показано в предната точка с вложени **if-else** конструкции, но резултатният програмен код става труден за четене и поддържане. За да улесни програмистите в такива ситуации, Java поддържа конструкцията **switch**, която позволява по-

прегледно структуриране на разклоненията на програмата. Тя има следния общ вид:

```
switch (променлива) {  
  
    case стойност_1:  
        ...  
        // програмен код, който се изпълнява в случай че променливата има за  
        // стойност стойност_1  
        ...  
        break;  
  
    case стойност_2:  
        ...  
        // програмен код, който се изпълнява в случай че променливата има за  
        // стойност стойност_2  
        ...  
        break;  
  
    ...  
  
    case стойност_n:  
        ...  
        // програмен код, който се изпълнява в случай че променливата има за  
        // стойност стойност_n  
        ...  
        break;  
  
    default:  
        ...  
        // програмен код  
}
```

Подобно на **if-else** конструкцията, **switch** започва със стойност в скоби, която следва непосредствено след ключовата дума, въвеждаща конструкцията. За разлика от **if-else**, стойността в скобите на **switch** типично е променлива, която *не* е от булев тип. Следователно променливата не може да има стойност **true** или **false** и въобще не може да бъде заменяна от логически проверки, които се използват при **if-else**.

Вместо да извършва разнообразни логически проверки, конструкцията **switch** проверява само за равенство между стойността на променливата в скоби и всяка от стойностите, зададени с ключовата дума **case**. Ако се установи равенство, кодът, който следва ключовата дума **case**, започва да се изпълнява.

Последователността на изпълнение на **switch** е демонстрирана в следващите няколко примера:

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.print("Въведете номер на месец от годината: ");
6         int month = new Scanner(System.in).nextInt();
7         switch (month) {
8             case 1:
9                 System.out.println("януари"); break;
10            case 2:
11                System.out.println("февруари"); break;
12            case 3:
13                System.out.println("март"); break;
14            case 4:
15                System.out.println("април"); break;
16            case 5:
17                System.out.println("май"); break;
18            case 6:
19                System.out.println("юни"); break;
20            case 7:
21                System.out.println("юли"); break;
22            case 8:
23                System.out.println("август"); break;
24            case 9:
25                System.out.println("септември"); break;
26            case 10:
27                System.out.println("октомври"); break;
28            case 11:
29                System.out.println("ноември"); break;
30            case 12:
31                System.out.println("декември"); break;
32            default:
33                System.out.println("Невалиден номер на месец");
34        }
35    }
36 }
```

Разпечатка 9. Превръщане на номер на месец в име на месец с използването на **switch**

Програмата на разпечатка 9 кара потребителя да въведе номер на месец, след което извежда името на месеца или съобщение за грешка, ако номерът, въведен от потребителя, не е между 1 и 12.

Ако потребителят въведе например 6, изпълнението на кода в **switch** конструкцията ще започне от ред 18, като кодът, отговарящ на секциите **case 1**, **case 2** и т.н. до **case 5** включително, ще бъде изпуснат.

Така първата декларация, която ще бъде изпълнена, е **System.out.println(“юни”)**. Тя коректно ще изведе името, което отговаря на въведения от потребителя номер на месец. Следващата декларация, която ще се изпълни, се състои от ключовата дума **break**. Тя ще прекъсне изпълнението на кода в **switch** конструкцията и програмата ще продължи от първата декларация, която следва тази конструкция. В примерната програма няма такава, което значи, че на този етап тя ще приключи работа.

Точно както в езиците C и C++, от които води началото си Java, изпълнението на кода в **switch** по подразбиране продължава последователно до достигане на края на декларацията. Тоест, ако на ред 18 нямаше **break**, изпълнението щеше да продължи с кода, предвиден за **case 7**. По подобен начин, при липса на **break** след декларациите за **case 7**, изпълнението щеше да продължи с **case 8** и т.н. Евентуално щяха да бъдат изпълнени дори декларациите, които се намират в секцията **default**.

C други думи, *ако липсваха* всички **break** декларации, програмата от разпечатка 9 щеше да изведе следното при задаване на шест за номер месеца:

```
Въведете номер на месец от годината: 6
юни
юли
август
септември
октомври
ноември
декември
Невалиден номер на месец
```

Това контраинтуитивно поведение на **switch** позволява неусетното вмъкване на грешки в програмата и вероятно се дължи на някои от недомислиците от ранните „детски“ години на C. Може би то щеше да бъде коригирано за изминалите 50 години, ако C не бе набрал толкова бързо популярност. Големият брой програми, написани на C, постави акцента върху съвместимостта и избягването на промени в езика, които биха повредили съществуващите програми.

Създателите на езика Java вероятно са се водили от същите съображения, когато са пренесли конструкцията от C без промени, позволявайки на множество програмисти на C и C++ максимално бързо да се прехвърлят към Java. Едва в последните години модерните версии на Java правят опит за корекция, като добавят нов синтаксис за **switch**. Този синтаксис ще бъде разгледан по-нататък.

Описаното поведение на **switch** не е винаги и изцяло негатив. Понякога то позволява креативно подреждане на стойностите и пести писането на програмен код. Програмата на следващата разпечатка е много подобна на предната, с тази разлика, че извежда сезона от годината, който отговаря на даден номер на месец.

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.print("Въведете номер на месец от годината: ");
6         int month = new Scanner(System.in).nextInt();
7         switch (month) {
8             case 12:
9             case 1:
10            case 2:
11                System.out.println("зима");
12                break;
13            case 3: case 4: case 5:
14                System.out.println("пролет");
15                break;
16            case 6:
17            case 7:
18            case 8:
19                System.out.println("лято");
20                break;
21            case 9:
22            case 10:
23            case 11:
24                System.out.println("есен");
25                break;
26            default:
27                System.out.println("Невалиден номер на месец");
28        }
29    }
30 }
```

Разпечатка 10. Превръщане на номер на месец в име на сезон с използването на **switch**

Ако потребителят стартира програмата от разпечатка 10 и въведе 6, изпълнението на декларациите ще започне от ред 16, където е изписан **case 6**.

След **case 6** обаче няма специални декларации или **break**. Това означава, че изпълнението ще премине към **case 7**. **case 7** също няма декларации или **break**, което означава, че изпълнението преминава към **case 8**. Там вече има програмен код, който отпечатва името на сезона (ред 19). На следващия ред има **break**, което предотвратява изпълнението на програмния код, предвиден за следващите сезони на годината.

С други думи, поведението на **switch** позволява определен брой случаи да бъдат групирани заедно и обработени с един и същи програмен код. Разбира се, читателите ще забележат, че разширеният нов синтаксис на **switch** постига това по-ефективно без да прибегва до контраинтуитивното поведение описано току-що.

Трябва да се обърне внимание на това, че в програмата от предната разпечатка, ние поставяхме **break** декларацията на същия ред като предхождащата я декларация. В настоящата програма **break** е на отделен ред. И двата варианта са допустими, тъй като за компилатора на Java по принцип няма значение дали всички декларации ще бъдат написани на един ред, стига да са правилно отделени една от друга с точка и запетая. По същия начин е допустимо, когато дадени **case** елементи нямат код, те да се изпишат на един ред, както сме направили с **case 3**, **case 4** и **case 5**. По принцип е добра практика, когато декларациите към даден **case** елемент са повече, те да се изпишат на отделни редове. Авторът предпочита да изписва **case** елементите без прилежащ код на отделен ред, като смята че това е по-четливо.

До момента не сме дискутирали предназначението на **default** частта в **switch** конструкцията. Тя е опционална и ако присъства, програмният код, асоцииран с нея, ще се изпълни когато за стойността на променливата в скобите не отговаря на нито една от стойностите, въведени с **case** елементите.

Така например ако потребителят въведе 13 или например -5 за номер на месец в един от двата последни примера, програмният код в **default** частта ще се изпълни и ще го уведоми, че е въвел невалиден номер на месец в годината.

На този етап можем да върнем читателите към разпечатка 7 и да обсъдим въпроса дали е възможно да реализираме същия пример (вложените **if-else** конструкции за определяне на оценка по ECTS) чрез **switch**. Отговорът е отрицателен. Конструкцията **switch-case** може да работи само с дискретни стойности, тоест задаването на интервали от рода на „2 до 3“ или изрази като $2 \parallel 3$ не е предвидено и не е допустимо. Освен това има съществени ограничения за типа на променливата/стойността в скобите на **switch** конструкцията. Тя може да бъде от тип **byte**, **short**, **int**, **char** и **String** или пък **enum** тип. Дробните типове (**float** и **double**), булевият тип (**boolean**), както и стойности от тип **long** не са позволени.

Особено внимание заслужава ситуацията, която в **switch** използваме **enum** типове, които още не са представени подробно в изложението. Долният пример илюстрира особеностите на тази ситуация.

```
1 enum Seasons {SPRING, SUMMER, AUTUMN, WINTER};
2
3 public class Main {
4     public static void main(String[] args) {
5         Seasons season = Seasons.WINTER;
6         switch(season) {
7             case SPRING:
8                 System.out.println("Дърветата цъфтят"); break;
9             case SUMMER:
10                System.out.println("Морето е топло"); break;
11        }
12    }
13 }
```

Разпечатка 11 **switch** със стойност от тип **enum**

Enum типовете са начин потребителят да дефинира свой тип, който изброява определен брой стойности. Променливите от този тип могат да приемат само тези стойности, което намаля шансовете за грешки. Така например сезоните могат вътрешно в програмата да се представят с такава **enum** стойност, както е показано на ред 1 от разпечатката. Това е по-добре от това например сезоните да се представят с цели числа вътрешно в програмата (някои цели числа биха били невалидна репрезентация за сезон и програмистът би следвало непрекъснато да

следи за ситуации, които биха поставили такава невалидна репрезентация в някоя променлива).

Стойностите на променливите от дадения **enum** тип типично се посочват навсякъде в програмата префиксирани с името на самия **enum**. Така например е направено със **Seasons.WINTER** на ред 5. Когато се подават като параметър на **case** елемент обаче, стойностите **не** се префиксират (редове 7 и 9 от разпечатката). Обратното не е валидна Java и ще произведе грешка при компилация.

Освен стандартния синтаксис, наследен от C и C++, Java 12 въвежда и нов, разширен синтаксис, който е по-близко до примитивите, внесени в езика за работа с ламбда функции. Този синтаксис е демонстриран в следващия пример:

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.print("Въведете номер на месец от годината: ");
6         int month = new Scanner(System.in).nextInt();
7         switch (month) {
8             case 12, 1, 2 -> {
9                 System.out.println("зима");
10                System.out.println("Времето не е подходящо за почивка на море");
11            }
12            case 3, 4, 5 -> System.out.println("пролет");
13            case 6, 7, 8 -> System.out.println("лято");
14            case 9, 10, 11 -> System.out.println("ноември");
15            default -> System.out.println("Невалиден номер на месец");
16        }
17    }
18 }
```

Разпечатка 12. Разширен синтаксис на *switch*

В разширения синтаксис на **switch** конструкцията знакът за двуеточие е заменен със стрелка надясно, формирана от символите „тире“ и „по-голямо“. Освен това разширеният синтаксис позволява изброяването на множество стойности в един **case** елемент, като отделните стойности са разделени със запетайка.

Най-важната промяна касае елиминирането на нуждата от **break** декларации. По подразбиране в разширения синтаксис на **switch**, на всеки **case** елемент отговаря само една декларация, както е показано на редове 12,13 и 14 на

разпечатката. Ако е нужно изпълнението на повече декларации, те могат да се оградят в блок от код, както е демонстрирано на редове 8-11.

3. Цикли с брояч

Както бе споменато в началото, многократното изпълнение на един и същи програмен код се случва изключително често в програмите по най-различни причини, най-важната от които е това, че програмите често обработват множество еднотипни обекти.

Циклите с брояч са най-използвания начин за реализиране на подобни повторения. Когато използваме такива цикли ние типично имаме точна идея колко пъти искаме да повторим изпълнението на даден сегмент от код.

Циклите с брояч в Java се задават с ключовата дума *for* и имат следния общ вид:

```
for (инициализатор; условие; брояч) {  
    тяло_на_цикъла  
}
```

Инициализаторът декларира и задава начална стойност на променливата, която ще играе роля на брояч на цикъла. Стойността на тази променлива участва в определянето на това колко пъти ще бъде изпълнено тялото на цикъла.

Отбелязаната като „**брояч**“ част на конструкцията всъщност представлява декларация, която задава нова, увеличена или намалена стойност на променливата-брояч. Тази декларация се изпълнява всеки път, когато завърши едно изпълнение на тялото на цикъла.

Условието пък представлява логически израз, който се изчислява преди всяко влизане в тялото на цикъла. Изразът се изчислява всеки път, когато предстои влизане в тялото на цикъла, включително и преди първото влизане. Ако изразът се изчисли до **true**, тялото на цикъла се изпълнява (още) веднъж. Ако се случи обратното – изразът – условие се изчисли до **false**, изпълнението на цикъла

се прекратява и изпълнението на програмата преминава към следващата след **for** декларация.

Описаното поведение е демонстрирано с един прост пример на следващата разпечатка:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         for (int i=0; i<10; i++) {  
4             System.out.println(i);  
5         }  
6     }  
7 }
```

Разпечатка 13. Цикъл с брояч

Цикълът от програмата на разпечатка 13 (редове 3-5) отпечатва числата от 0 до 9, всяко на отделен ред. Много е полезно, особено за читателите, които се сблъскват за първи път с подобна езикова конструкция, да бъде обяснено постъпково изпълнението на горната програма.

При влизането в цикъла на ред 3 се изпълнява *инициализаторът*, който е първото от трите „полета“, разделени с точка и запетая в скобите на ред 3, а именно **int i=0**; Това води до дефинирането на променливата **i**, която ще бъде брояч на цикъла. Инициализаторът задава за начална стойност на променливата числото 0.

Следващото действие е да се провери условието за влизане в тялото на цикъла, т.е. второто поле, а именно **i<10**. Тъй като стойността на **i** към този момент е 0, **i<10** е вярно и изразът се изчислява до **true**. Това означава, че тялото на цикъла ще бъде изпълнено веднъж.

Самото тяло на цикъла (без да броим отварящата и затварящата фигурна скоба на блока) е разположено на един ред – ред 4. То опечатва стойността на променливата брояч. Така при първото изпълнение на тялото на цикъла, отпечатаната стойност ще бъде 0.

На този етап изпълнението тялото на цикъла е приключено. Следващото действие, което ще бъде изпълнено, е инструкцията за увеличаване на брояча

(третото „поле“ на ред 3). Тази инструкция всъщност може да бъде всякакъв израз, който присвоява стойност на брояча и/или други променливи, но типично е „формула“, която увеличава стойността на брояча, най-често със стъпка от 1. Случаят в примера е точно такъв и `i++` води до увеличаването на променливата брояч `i` с едно. Така новата стойност на променливата – брояч става 1.

На този етап ще се извършат действията, които определят дали повторението на тялото на цикъла ще бъде изпълнено отново или пък, евентуално, дали цикълът ще бъде прекратен.

Както бе споменато по-рано, инициализаторът се изпълнява само веднъж, в самото начало на цикъла. Този път той няма да се изпълни, а направо ще се премине към проверката за влизане в цикъла. `i<10` отново е вярно и дава **true**, тъй като `i` е 1. Това значи, че тялото на цикъла ще се изпълни отново. Подобно еднократно повторение на цикъла се нарича „итерация“.

Итерациите ще продължат по подобен начин за `i` равно на 2, `i` равно на 3 и т.н. докато евентуално ще завършим итерация за `i` равно на 9. В края на тялото на цикъла ще се изпълни инструкцията за увеличение `i++` и `i` ще стане 10.

Преди да се премине към следващо изпълнение на тялото на цикъла, ще бъде направена проверката `i<10`. Тя обаче ще произведе **false**, тъй като `i` вече е равно на 10 и `10<10` не е вярно. На този етап изпълнението на цикъла ще се прекрати като последната отпечатана стойност ще бъде 9.

След цикъла няма друга инструкция, което значи че методът **main** и програмата завършват.

Практиката показва, че повечето начинаещи изпитват проблеми, когато трябва да формулират трите „полета“ на циклите с брояч на база изискванията на реалната задача. Това се дължи на тяхната особена структура, която позволява голяма гъвкавост за сметка на минимално семантично усложняване на синтаксиса. Така например при **for** ние не говорим просто за начална стойност и крайна стойност на брояча, нито за стъпка на увеличение на този брояч. Вместо това имаме „поле“ за деклариране на променлива. Това поле може да се използва

за деклариране на нова променлива, но може и да се използва само за задаване на начална стойност на вече съществуваща променлива, която ще играе роля на брояч на цикъла. Декларацията дори може да се изпусне.

По същия начин полето с инструкции за увеличение на брояча всъщност позволява всякаква декларация, която присвоява следващата стойност на променливата – брояч. Това поле също може да се изпусне напълно, като в този случай увеличението на променливата – брояч ще трябва да се извърши ръчно в тялото на цикъла.

Най-проблемно за усвояване от начинаещи е полето за проверка при влизане в цикъла. То се изчислява винаги, дори при първото влизане в цикъла. Цикълът се изпълнява само ако проверката върне **true**. При връщане на **false**, цикълът се прекратява. Това дава гъвкавост и позволява формулиране на „условие за прекратяване“ както на цикли с увеличаващ се брояч, така и на цикли с намаляващ брояч, както и на цикли, при които прекратяването на цикъла зависи от някакво сложно изчисление, в което участва променливата – брояч.

Въпреки това повечето цикли не са толкова сложни, поради което е полезно начинаещите да мислят в понятията начална стойност – крайна стойност и стъпка на увеличение.

В нашия случай искаме да започнем извеждането от 0 и да приключим на 9 включително. Това значи, че началната стойност на нашия брояч трябва да бъде 0, което дава инициализатора **i=0**. Тъй като крайната стойност е по-голяма от началната стойност, нашият брояч трябва да се увеличава. При това увеличението трябва да е със стъпка едно, което дава инструкция за увеличаване на брояча **i++**. Последното нещо, което трябва да бъде формулирано е условието за влизане в цикъла. Тъй като 9 е крайна стойност, е ясно че 9 ще фигурира в това условие. Също така е ясно, че променливата брояч също ще бъде част от условието. Това дава два основни варианта **i<9** или **i>9**. Ако искаме да обхванем крайния случай, тоест да изпълним последно тялото на цикъла със стойност за **i**, равна на 9, ние

можем да напишем **i<=9** или **i>=9**. Последните две обаче са еквивалентни съответно на **i<10** и **i>10**.

На този етап читателят трябва да избере един от двата варианта и е полезно да се подсети, че проверката трябва да е така формулирана, че да дава **true** когато цикълът трябва да продължи (броячът е между началната и крайната стойност) и **false** в първия момент, когато броячът надхвърли границата. Това дава, разбира се, **i<=9** или неговия еквивалент **i<10**.

Когато броячът се увеличава, това означава, че ще се избере операторът **<** или **<=**. В противен случай, ако броячът намалява от голяма начална стойност към малка крайна стойност, трябва да се избере операторът **>** или **>=**.

Илюстрираме казаното с пример, който отпечатва всички четни числа между 20 и 1 включително в обратен ред:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         for (int count=20; count>=1; count-=2) {  
4             System.out.println(count);  
5         }  
6     }  
7 }
```

Разпечатка 14. Цикъл с брояч

В програмата от цикъл 14, променливата брояч започва от висока стойност и върви надолу, като инструкцията за получаване на нова стойност на брояча гласи **count-=2**. Това означава, че променливата **count** последователно ще приеме стойности 20, 18, 16, 14, 10 и т.н. Процесът ще спре когато изразът, който определя дали цикъла продължава, спре да дава като резултат **true**. При тези условия търсим зависимост между **count** и 1 която е от тип **>=**.

Последната итерация на цикъла ще бъде при **count** равно на 2. Следващата стойност на брояча ще бъде 0, но стойност 0 за **count** ще направи **count>=1** невярно и изчислението му ще даде резултат **false**, което ще прекрати цикъла.

Понякога се налага дадена итерация или дори целият цикъл да бъдат прекратени предсрочно. Това се извършва с ключовите думи **break** и **continue**. Ключовата дума **break** прекратява целия цикъл и изпълнението на програмата продължава от следващата след цикъла декларация. Ключовата дума **continue** прекратява изпълнението *само* на тялото на цикъла. Тоест, при достигане на **continue**, изпълнението прескача оставащите декларации до края на блока от код, който представлява тялото на цикъла. След това изпълнението продължава като при нормален цикъл – изпълнява се първо инструкцията за увеличаване на променливата брояч, след което се изпълнява проверката за влизане в тялото на цикъла.

Действието на **break** и **continue** е демонстрирано на следващите две разпечатки.

```
1  /*
2  * Намира най-голямото цяло положително число, което е по-малко
3  * от 1000 и се дели точно на 17.
4  */
5
6  public class Main {
7      public static void main(String[] args) {
8          for (int i=1000; i>=0; i--) {
9              if (i % 17 == 0) {
10                 System.out.println(i);
11                 break;    // Изпълнението продължава от ред 14
12             }
13         }
14     }
15 }
```

Разпечатка 15. Използване на **break** за прекратяване на цикъл с брояч

Цикълът на разпечатка 15 последователно присвоява на променливата – брояч стойностите от 1000 до 0 включително. Ако някоя от тези стойности се дели на 17, това е най-голямата стойност, която се дели точно на 17 и е по-малка или равна на 1000. Декларацията **if** на ред 9 проверява точно това.

Ако остатъкът при делението на **i** на 17 (**i%17**) е равен на 0, ние сме намерили нужната стойност и няма смисъл да продължаваме изпълнението на цикъла. В такъв случай извеждаме стойността и прекратяваме изпълнението на цикъла с **break** (редове 10 и 11).

```

1  /*
2  * Отпечатва всички числа от 1 до 100 включително, които не се
3  * делят точно на 3.
4  */
5
6  public class Main {
7      public static void main(String[] args) {
8          for (int i=1; i<=100; i++) {
9              if (i%3 == 0) {
10                 continue; // Изпълнението продължава на ред 8
11             }
12             System.out.println(i);
13         }
14     }
15 }

```

Разпечатка 16. Използване на **continue** за прекратяване на една итерация на цикъл с брояч

Програмата на разпечатка 16 присвоява на променливата брояч **i** последователно стойностите от 1 до 100 включително. **if** конструкцията на редове 9-11 проверява дали **i** се дели точно на 3 и ако е така, прекратява предстрочно изпълнението на *тялото* на цикъла, преди то да достигне ред 12, където стойността се отпечатва. По този начин за всяко число, което се дели на 3, отпечатването бива пропуснато.

Завършваме дискусията на циклите с брояч, обръщайки внимание на два важни факта. Първо, дори опитните програмисти допускат грешки при формулиране на ограниченията на циклите с брояч. Най-честата грешка касае извършването (или неизвършването) на последната итерация. Такива грешки се наричат често „**off-by-one**“, т.е. приблизително на български „с едно повече / по-малко“. Второ, променливата брояч на много цикли не участва директно в изчисленията за нещо друго освен за следене на поредния номер на елемента, върху който се прилага дадено действие.

За подобни ситуации Java предлага „**for-each**” (буквално „за-всеки“) конструкция, която позволява да се обхождат всички елементи в дадена съвкупност без да се използва променлива – брояч, като по този начин се избегне възможността да се допуснат грешки.

Долният пример илюстрира това и е поместен тук, макар че към момента нямаме достатъчно знания за структурите в Java, които съдържат множество еднотипни елементи (масиви и колекции).

```
1 import java.util.List;
2
3 public class Main {
4     public static void main(String[] args) {
5         var gradove = List.of("София", "Варна", "Свищов");
6         for (String grad : gradove) {
7             System.out.println(grad); //Отпечатва името на града
8         }
9     }
10 }
```

Разпечатка 17. **for-each** конструкция

Програмата от разпечатка 17 създава списък от еднотипни елементи, които в случая са символни низове, представляващи имена на градове (ред 5). Този списък има 3 елемента – „София“, „Варна“ и „Свищов“. Цикълът на редове 6-8 много би приличал на циклите, демонстрирани по-горе, ако имаше променлива брояч. Такава няма и **String grad : gradove** всъщност означава че променливата **grad** последователно ще приеме всяка една от стойностите в списъка. Така тялото на цикъла ще се изпълни три пъти – по един за всеки елемент. Повече детайли за **for-each** конструкцията ще бъдат дадени когато изложението достигне до масивите и колекциите в Java.

Вторият важен детайл касае обхвата и видимостта на променливите в даден **for** цикъл и е илюстриран със следния пример:

```
1 public class Main {
2     public static void main(String[] args) {
3         int counter = 5;
4         for (; counter<10;) {
5             int i = counter*2 + counter;
6             System.out.println(counter);
7             counter++;
8         }
9         System.out.println(counter);
10        System.out.println(i); //Грешка при компилация
11    }
12 }
```

Разпечатка 18. Видимост на променливите при цикъл с брояч

За читателите, които тепърва са се запознали с писането на цикли с брояч, горната програма ще изглежда много странна. Така например цикълът на ред 4 има празни полета за инициализатор и увеличение на брояча. Това е напълно допустимо и илюстрира казаното по-горе - конструкцията **for** по замисъл и начин на задаване на параметрите е много гъвкава. Вместо като параметър на цикъла, инструкцията за увеличение на брояча е дадена като отделна декларация на ред 7.

Възможността променливата брояч да се променя по този начин дава гъвкавост и свобода на програмиста. Така например той може да формулира правила за брояча, които не увеличават или намаляват променливата с равни стъпки и дори правила, които движат брояча в различни посоки на всяка итерация. Начинаещите програмисти трябва да помнят обаче, че за прости задачи като горния пример е по-добър стил на програмиране увеличението на брояча да се декларира в скобите на конструкцията **for**.

Далеч по-важното наблюдение, което трябва да се направи върху програмата от примера е това, че променливата брояч всъщност е декларирана преди началото на цикъла.

Възможността за брояч да се използва вече съществуваща променлива също дава гъвкавост. Така началната стойност на брояча може да не е фиксирана, а да се определя от резултата от даден процес или изпълнението на даден алгоритъм, който предхожда дадения цикъл.

Което е по-важно, декларирането на променливата брояч преди цикъла ни позволява да достъпим нейната последна стойност след края на цикъла, което е особено полезно, ако прекъснем цикъла с **break** или **continue**.

В противовес, ако променливата е декларирана в блока от код в тялото на цикъла или пък в инициализатора, тя ще престане да съществува с приключването на цикъла. Редове 5 и 10 демонстрират точно това – променливата **i** е декларирана в тялото на цикъла и съществува в рамките на отделна итерация.

На ред 10 променливата вече не съществува, което поражда грешката при компилация.

4. Цикли с **while** и **do-while**

Понякога броят повторения на даден цикъл не е фиксиран и зависи от настъпването на дадено условие, което може да се определи само по време на изпълнение на програмата. Подобни цикли не могат да се моделират особено елегантно с помощта на конструкциите за цикли с брояч и за тях е нужна по-обща и универсална конструкция. Циклите с конструкциите **while** и **do-while** играят точно такава роля в Java.

Веднъж щом познаваме конструкцията **for**, ние можем да мислим за тези две конструкции като за „минималистичен“ **for**. Те нямат механизми за работа с брояч, но все още проверяват по същия начин логическо условие за влизане в тялото на цикъла, което трябва да е **true** за да се изпълни както първата, така и всяка последваща итерация.

Конструкциите **while** и **do-while** имат следния общ вид:

```
while (условие) {  
    тяло_на_цикъла  
};
```

```
do {  
    тяло_на_цикъла  
} while (условие);
```

Тялото на цикъла и при двете конструкции се повтаря, ако логическото условие се изчисли до **true** и се прекратява, ако логическото условие съответно стане **false**. Разликата между двете конструкции се корени в това кога се проверява логическото условие. При **while**, подобно на досега разгледаните цикли с **for**, логическото условие се проверява преди влизане в цикъла, което означава, че е възможно тялото на цикъла да не се изпълни нито веднъж.

При **do...while** логическото условие се проверява в края на цикъла, което означава, че във всички случаи ще се изпълни поне една итерация на цикъла.

Долният пример демонстрира използването на **do...while**:

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String... args) {
5         Scanner scanner = new Scanner(System.in);
6         int a;
7         do {
8             System.out.println("Въведете четно число: ");
9             a = scanner.nextInt();
10        } while (a%2!=0);
11    }
12 }
```

Разпечатка 19. Демонстрация на **do...while** конструкция

Програмата на разпечатка 19 пита потребителя за четно число. Тъй като **do...while** цикълът се изпълнява поне веднъж, програмата ще изпълни редове 8 и 9 преди да извърши проверката за повторение на цикъла. Така потребителят ще може да въведе число. След като числото е въведено, тялото на цикъла приключва и идва ред на логическата проверка, която по същество проверява дали числото е *нечетно* (по-конкретно дали остатъкът при деление на това число на две не е равен на нула). Ако числото е нечетно, цикълът ще се повтори и потребителят ще бъде запитан отново за четно число. В крайна сметка цикълът на редове 7-10 ще завърши тогава, когато потребителят въведе в променливата **a** четно число.

Разпечатка 20 демонстрира използването на **while** цикъл.

```
1 public class Main {
2     public static void main(String... args) {
3         int a = 26624;
4         while (a%2==0) {
5             System.out.println(a + " се дели на 2");
6             a=a/2;
7         }
8         System.out.println(a + " не се дели на 2");
9     }
10 }
```

Разпечатка 20. Демонстрация на **while** конструкция

Програмата на разпечатката използва **while** цикъл, за да провери колко пъти дадено число може да бъде разделено на 2. При така зададената стойност на **a**, на първото изпълнение на ред 4 логическата проверка ще даде **true**, тъй като стойността в **a** е четна. Това ще доведе до изпълнението на тялото на цикъла, което ще изведе информация на потребителя, че числото се дели на 2 и ще извърши реалното разделяне на стойността в **a** на две. С това първата итерация на цикъла приключва. Ако новото число в **a** отново се дели на 2, цикълът ще се повтори отново и така докато в **a** евентуално се окаже число, което не се дели на 2. Тогава проверката на ред 4 ще даде резултат **false** и изпълнението на програмата ще продължи от реда след **while** цикъла, а именно ред 8.

Глава 5. Работа със символни низове

Символните низове представляват последователности от букви, цифри и/или други символи. В компютърните програми ние най-често използваме символните низове за съхраняване на текстова информация, предназначена за четене от потребителя. Така например като символни низове се съхраняват имената на бутоните и менютата в потребителския интерфейс на програмите, съобщенията за грешка, текстът на генерираните от програмата уеб-страници и пр. Също като символни низове се съхраняват подканянията и съобщенията в диалоговите прозорци. Текстови низове също така са и имената на хора, градове, държави, месеци, фирми, учреждения, отдели и т.н.

Изброените примери далеч не изчерпват множеството ситуации, в които ние представяме данните като символен низ. При нужда и желание ние можем да представим като символни низове почти всички данни, с които работят програмите, включително числа, дати, булеви стойности и пр.⁹

Символните низове на практика се използват навсякъде и далеч не е изненадващо, че тяхното представяне и средствата за работата с тях са важни и определящи елементи на всеки един език за програмиране.

1. Символни променливи и символни литерали

Точно като при другите типове данни, представени досега, ние можем да имаме както литерали, така и променливи от тип символен низ.

Символните литерали, също като литералите от другите елементарни типове, са просто фиксирани символни низове, които са вписани директно в

⁹ Трябва да се има предвид, че използването на символни низове за съхраняване на данни, които могат да се съхранят в друг специализиран тип означава, че се отказваме от удобствата, които ни носи използването на този специализиран тип. Например нищо не пречи да съхраняваме числа като символни низове. Но тогава тези „числа“ няма да могат да се събират, изваждат, умножават и делят лесно с вградените в Java оператори.

програмата още по време на нейното създаване. Такива литерали в програмата се изписват оградени в двойни кавички.

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String... args) {
5         System.out.println("Това е символен литерал");
6         System.out.println("Въведете номер на ден от седмицата [1-7]: ");
7         switch (new Scanner(System.in).nextLine()) {
8             case "1": System.out.println("Понеделник"); break;
9             case "2": System.out.println("Вторник"); break;
10            case "3": System.out.println("Сряда"); break;
11            case "4": System.out.println("Четвъртък"); break;
12            case "5": System.out.println("Петък"); break;
13            case "6": System.out.println("Събота"); break;
14            case "7": System.out.println("Неделя"); break;
15            default: System.out.println("Въвели сте невалиден номер на ден от
седмицата!");
16        }
17    }
18 }
```

Разпечатка 1. Програма, демонстрираща изписването и използването на символни литерали в Java

Програмата на разпечатка 1 има символни литерали на почти всеки ред. Първият от литерал се намира на ред 5 и гласи „Това е символен литерал“. Този литерал (както впрочем и всички други литерали в тази и всички останали програми) е фиксиран и вписан от програмиста още при писането на програмата.

На ред 5 литералът е подаден като параметър на метода **System.out.println**. Така изпълнението на ред 5 ще доведе до това, че текстът „Това е символен литерал“ ще бъде изведен от програмата.

Освен на ред 5, символни литерали има и на редове 6 и 8 до 15 включително, като наблюдателните читатели ще забележат, че редовете, които започват с `case` имат по два.

В Java **променливи, които съхраняват символни низове, се декларират чрез типа String**, както е показано в програмата на разпечатка 2.

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         String versija = "Версия 1.0";
6         String name;
```

```

7      String surname;
8
9      Scanner s = new Scanner(System.in);
10     System.out.print("Въведете име: ");
11     name = s.nextLine();
12     System.out.print("Въведете фамилия: ");
13     surname = s.nextLine();
14     String fullName = name + " " + surname;
15     System.out.println(fullName);
16 }
17 }

```

Разпечатка 2. Програма, демонстрираща декларирането на променливи от тип **String**

В програмата на разпечатка 2 ние декларираме няколко променливи от тип **String**, предназначени за съхраняване на символни низове. Също като променливите от елементарни типове, дискутирани по-рано, декларацията на променлива от тип символен низ се състои от ключова дума, указваща типа (тоест ключовата дума **String**), следвана от името на променливата и евентуална начална стойност.

На ред 5 е демонстрирано декларирането на променлива от тип **String** със задаване на начална стойност, докато на редове 6 и 7 са показани декларации на променливи без начална стойност.

На ред 9 програмистът създава специален обект, който позволява въвеждане на стойности от системната конзола¹⁰, а на редове 11 и 13 той използва този обект за да позволи на потребителя да въведе стойности на двете променливи – name и surname, които бяха декларирани на редове 6 и 7 без начална стойност.

Най-интересният ред на програмата е ред 14, който демонстрира, че операторът **+** може да се използва и върху променливи от тип **String**. При символните низове обаче този оператор има различно действие и води до „слепването“ на техните стойности.

¹⁰ По исторически причини текстовите приложения използват понятието „конзола“ за обозначаване на комбинацията от екрана и клавиатурата на компютъра.

Ред 14 ще вземе стойността, въведена от потребителя във променливата `name`, ще долепи към нея символния литерал " ", който се състои от един празен интервал, след което ще долепи и стойността, която потребителят е въвел в променливата `surname`. Като резултат ще се получи стойност, която представлява пълното името и фамилията на потребителя, разделени с интервал. Тази нова стойност ще бъде присвоена на променливата `fullName` и ще бъде изведена на конзолата с извикването на метода на следващия ред 15.

2. Методи за работа със символни низове

За разлика от елементарните типове, разгледани до момента, типът `String` всъщност е клас, което означава, че променливите от тип **`String`** имат специализирани методи. Извикването на тези методи предизвиква извършването на специфично действие върху съдържанието на променливата. Като правило, това действие произвежда нова стойност от тип **`String`**.

Начинът на работа с методите на класа **`String`** е демонстриран в следващия пример:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         String helloWorld = "Hello, World!";  
4         System.out.println(helloWorld.length());  
5         System.out.println(helloWorld.toUpperCase());  
6         System.out.println(helloWorld  
7             .substring(7)  
8             .toUpperCase());  
9         System.out.println("    Hello    "  
10            .trim()  
11            .toUpperCase()+  
12            "world".toUpperCase());  
13     }  
14 }
```

Разпечатка 3. Извикване на методи на класа **`String`**

Програмата ще изведе следния изход:

```
13  
HELLO, WORLD!  
WORLD!  
HELLOWORLD
```

Първото и най-важно нещо, което читателите трябва да забележат в разпечатка 3, е това как се извикват методите на класа **String**. Първо се изписва името на *променлива* от този тип. Именно върху стойността на тази променлива ще приложим метода. Веднага след това се изписва точка, следвана от името на метода плюс евентуалните му параметри, заградени в скоби.

Така например на ред 4, потребителят извиква метода **length** на класа **String**. Този метод не взема параметри (поради което след него се изписват празни кръгли скоби) и връща като резултат дължината на символния низ.

По-наблюдателните читатели ще забележат, че низът, който ще „премерим“ по този начин, не е посочен като параметър. Това е така, тъй като низът всъщност е стойността на променливата **helloWorld**. Нашият запис (т.е. **helloWorld.length()**) означава точно това – “прилагаме” метода **length** към стойността на променливата **helloWorld**.

Подобен синтаксис е типичен за обектно-ориентираните езици като Java. Тези езици позволяват на потребителя да дефинира собствени „съставни“ типове, наречени „класове“. Класовете могат да имат разписани от потребителя стандартни „действия“ (т.е. „методи“), които се прилагат върху променливите от този тип. Методът **length** е именно такъв метод на класа **String**.

От съображения за краткост напред ние ще казваме че „извикваме“ метода **length** на **helloWorld** вместо „прилагаме метода **length()** на класа **String** върху променливата **helloWorld**“.

Резултатът от извикването на **helloWorld.length** е 13, тъй като символният низ в **helloWorld** е точно 13 символа. Това се подава като параметър на метода **System.out.println**, който отпечатва тази стойност на конзолата.

Методът **toUpperCase** превръща дадения символен низ в главни букви. Ето защо ред 5 извежда „HELLO, WORLD!“. Важно е да се отбележи, че този метод произвежда нов символен низ, тоест извикването на **toUpperCase** няма да предизвика промяна на стойността на променливата **helloWorld**.

Далеч по-интересни са редове 6 до 12 на разпечатката. Редове 6-8 демонстрират полезно следствие от това, че методите на класа **String** връщат нова стойност от тип символен низ. Това позволява множество от тях да се свързват заедно за постигане на по-сложни трансформации.

Извикването на метода `substring` с параметър 7 върху символния низ **helloWorld** създава нов низ, който съдържа символите от седма позиция на оригиналния низ до неговия край. С други думи **helloWorld.substring(7)** връща символния низ „World!“. Но върху този нов стринг е извикан още един метод – **toUpperCase**. Този нов метод превръща всички букви на „World!“ в главни, като не променя не-буквените символи. Така новият низ е „WORLD!“. Именно той се извежда на конзолата.

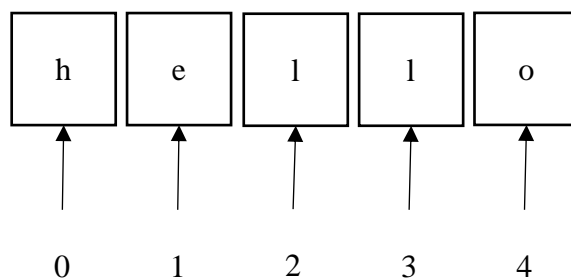
Редове 9-13 демонстрират още една особеност – методите на класа `String` могат да се извикват даже върху символни литерали. Така методът `trim` на ред 10 ще се изпълни върху литерала „ Hello “ и ще създаде нов символен низ, от който ще бъдат премахнати началните и крайните интервали.

По същия начин върху литерала „world“ на ред 12 ще бъде приложен методът **toUpperCase**, което ще създаде нов символен низ със съдържание „WORLD“.

2.1. Номериране на позициите в символните низове

Много от методите, които работят с низове оперират с части от тях. По тази причина програмистите често се налага да задават определени позиции като параметри.

Полезно е да се знае, че номерирането на символите в символните низове започва от 0.



При методите, които вземат за параметър позиция в низа, подаването на позиция извън низа често генерира изключение, което може да бъде обработено с **try .. catch** конструкция. Съществуват и методи, при което това подаването на позиция извън низа е напълно приемливо и не пречи на нормалната работа на метода.

2.2. Методи **length** и **isEmpty**

Методите **length** и **isEmpty** са дефинирани по следния начин:

```
int length()
boolean isEmpty()
```

Методът **length** връща броя на символите в символния низ. Особено често се налага да се проверява дали даден символен низ е празен, т.е. има 0 символа. Това може да се провери с използването на **length**, но за по-голямо удобство, Java предлага метода **isEmpty**, който връща **true** ако символният низ е празен и **false** ако символният низ съдържа символи.

Долната програма демонстрира използването на двата метода:

```
1 public class Main {
2     public static void main(String[] args) {
3         String none = "";           // празен символен низ
4         String some = "abc";
5         System.out.println(none.length()); // Извежда 0
6         System.out.println(none.isEmpty()); // Извежда true
7         System.out.println(some.length()); // Извежда 3
8         System.out.println(some.isEmpty()); // Извежда false
9     }
10 }
```

2.3. Методи charAt, indexOf и lastIndexOf

Методите **charAt**, **indexOf** и **lastIndexOf** са дефинирани по следния начин, като последните два метода имат множество алтернативни версии:

```
char charAt(int index)
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

Методът **charAt** връща символа, който се намира на указаната от **index** позиция. Връщаната стойност е от тип **char**. Ако **index** е отрицателно число или указва позиция, която се намира след края на низа, Java ще генерира изключение от тип **IndexOutOfBoundsException**.

Методът **indexOf** има не по-малко от четири версии. Първите две в горната таблица позволяват на разработчика да търси *първата позиция*, на която се намира даден символ **ch** в дадения символен низ. Ако потребителят е задал параметъра **fromIndex**, търсенето ще започне от указаната позиция и ще продължи до края на низа. В противен случай ще бъде претърсен целият символен низ от първия до последния символ.

Търсенето на първата позиция, на която се среща даден символ в даден символен низ, без съмнение е полезно, но много често се налага да се търси не само един конкретен символ, а цяла последователност от символи, т.е. друг символен низ. В такива случаи казваме, че търсим позицията на „подниз“ в дадения символен низ.

Третата и четвъртата вариация на метода **indexOf** търсят именно позицията на подниз **str**, като отново позволяват на програмиста да търси подниза

от определена позиция (**fromIndex**) нататък или пък търсенето да обхване целия претърсван символен низ. Също като при първите два метода, върнатата стойност ще бъде първата позиция, на която се среща даден подниз в претърсвания низ (отчитайки че при задаването на **fromIndex** търсенето започва от указаната позиция нататък).

Ако не намерят търсения символ или подниз, четирите функции ще върнат -1. Ако програмистът по грешка подаде за **fromIndex** позиция след края на низа това няма да генерира изключение и функцията ще завърши нормално с върната стойност -1.

Методите **lastIndexOf** са аналогични на разгледаните 4 метода **indexOf**, с тази разлика, че те търсят последната позиция, на която се среща определен символ или подниз в претърсвания низ.

```
1 public class Main {
2     public static void main(String[] args) {
3         String name = "Viktor";
4         System.out.println(name.charAt(0));    // Извежда "V"
5         System.out.println(name.indexOf("i")); // Извежда "1"
6         System.out.println("Elina".indexOf('L')); // Извежда "-1"
7         System.out.println("voodoo".lastIndexOf('o'));
8                                                     // Извежда "5"
9         System.out.println("abc".charAt(3));
10                                                     // IndexOutOfBoundsException
11     }
12 }
```

Разпечатка 6. Методи **charAt**, **indexOf** и **lastIndexOf** на **String**

2.4. Методи **toLowerCase** и **toUpperCase**

Методите **toLowerCase** и **toUpperCase** са дефинирани по следния начин:

```
String toLowerCase()
String toUpperCase()
```

Методът **toUpperCase** копира оригиналния символен низ в нов символен низ, като замества всички малки букви от оригиналния низ са заменени с главни. По същия начин **toLowerCase** създава нов символен низ, в който всички букви от

оригиналния низ са заменени с малки. И двата метода манипулират само буквени символи. Небуквените символи се копират без промяна.

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("Hello world!"
4             .toLowerCase()); // Извежда "hello, world!"
5         System.out.println("Hello world!"
6             .toUpperCase()); // Извежда "HELLO, WORLD!"
7     }
8 }
```

Разпечатка 7. Методи toLowerCase, toUpperCase на String

2.5. Методи startsWith и endsWith

Методите **startsWith** и **endsWith** са дефинирани по следния начин:

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

Методът **startsWith** връща **true**, ако низът започва с подниза **prefix** и **false** в противен случай. Методът **endsWith** е аналогичен, но проверява дали низът завършва с указания суфикс.

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("test".startsWith("te")); //true
4         System.out.println("test".startsWith("Te")); //false
5         System.out.println("test ".endsWith("t")); //false
6         System.out.println("test ".endsWith("t ")); //true
7     }
8 }
```

Разпечатка 8. Демонстрация на методите **startsWith** и **endsWith** на String

Програмата на разпечатка 8 демонстрира използването на **startsWith** и **endsWith** за определянето на това дали даден низ започва по определен начин. Ред 3 ще отпечата “true”, тъй като думата „test“ наистина започва с „te”. Ред 4 ще отпечата “false”, тъй като Java прави разлика между малки и големи букви.

Изпълнението на ред 5 на програмата ще предизвика извеждането на “false”, тъй като низът, върху който се извиква **endsWith**, всъщност завършва с интервал, докато параметърът на **endsWith** съдържа само буквата “t”. Това

показва, че интервалите се третират от двете функции „равноправно“, т.е. по същия начин като останалите символи.

На ред 6, проверката вече пита дали низът завършва със символ “t”, следван от интервал. Това е така, поради което резултатът от изпълнението на реда е извеждането на “true”.

2.6. Методи trim, strip, stripLeading и stripTrailing

Методите **trim**, **strip**, **stripLeading** и **stripTrailing** са дефинирани по следния начин:

```
String trim()  
String strip()  
String stripLeading()  
String stripTrailing()
```

Методите могат да се използват за премахване на „излишни“ интервали, намиращи се в началото и края на символен низ. Такива символни низове и действието на методите върху тях са демонстрирани в следващия пример.

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("a "+  
4             "    star    ".trim()+  
5             " t    ".strip()); // Отпечатва "a start"  
6  
7         System.out.println("    is".stripLeading()+  
8             " born".stripTrailing()); // Отпечатва "is born"  
9     }  
10 }
```

Разпечатка 9. Демонстрация на методите **trim**, **strip**, **stripLeading** и **stripTrailing** на класа **String**

Докато **trim** е класическият начин за премахване на интервали от символните низове, трябва да се има предвид, че с развитието на Unicode стандарта в него се появиха множество символи, които не са интервал, но също като интервала обозначават „празно“ място. Затова Java версия 11 въведе трите други метода.

Методът **strip** премахва както интервали, така и други символи, които се интерпретират като празно място и се намират на в началото и в края на низа. Другите два варианта на този метод – **stripLeading** и **stripTrailing** – премахват съответно „празните“ символи, ако се намират само в началото или само в края на символния низ.

2.7. Метод substring

Методът **substring** е дефиниран в два варианта:

```
String substring(int beginIndex)
String substring(int beginIndex, endIndex)
```

Методът извлича част от символния низ като нов низ. Ако програмистът зададе само **beginIndex**, новият символен низ ще съдържа копие на символите в оригиналния низ, намиращи се от позиция **beginIndex** включително до края на низа.

Ако програмистът посочи и **endIndex**, тогава новият низ ще съдържа символите на оригиналния низ, които се намират на позиции между **beginIndex** включително и **endIndex**, без да включва символа на позиция **endIndex**.

Ако **beginIndex** е отрицателно число или **endIndex** е след края на низа, или пък ако **beginIndex** е по-голяма стойност от **endIndex**, функциите ще генерират изключение **IndexOutOfBoundsException**.

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("minimax".substring(4)); // max
4         System.out.println("minmax".substring(0,3)); // min
5     }
6 }
```

Разпечатка 10. Демонстрация на метода **substring**

2.8. Метод split

Методът **split** позволява разделянето на символен низ на отделни части и е дефиниран по следния начин:

<pre>String[] split(String regex, int limit) String[] split(String regex)</pre>
--

Методът **split** търси в оригиналния низ разделител, зададен чрез параметъра **regex**. Всеки от поднизове, който се намира между тези разделители и края на низа се поставя в масив, който е и стойността, връщана от метода.

Методът има две вариации. Ако програмистът не зададе ограничение чрез **limit** или пък **limit** е зададен, но има стойност 0, низът ще бъде разделен на толкова части, колкото предполагат наличните разделители. Ако в края на низа има празни поднизове, т.е. множество последователни копия на разделителя без друго съдържание между тях, тези празни поднизове няма да бъдат поставени в резултатния масив.

Ако обаче **limit** е зададен и стойността му е позитивна, тогава тази стойност се интерпретира като максимален брой на частите, които трябва да се получат след разделянето. Ако има повече от необходимите разделители, те няма да се вземат предвид и оставащата част от оригиналния низ, включително излишните разделители ще бъдат добавени към последния подниз, връщан в масива с резултатни стойности. Типично програмист би използвал подобен подход само ако е критично връщаният масив да е с предварително известна големина.

Ако **limit** е негативен, разделянето ще се извърши на толкова части, колкото предполагат разделителите. При това ако в края на низа има множество празни поднизове, разделени с разделителя (т.е. множество копия на разделителя едно след друго, без съдържание помежду тях), те също ще бъдат добавени към резултатния масив.

Докато обясненията, дадени до момента, изглеждат изчерпателни, всъщност това не е така. Причината за това е, че разделителят при метода **split** се интерпретира като регулярен израз. Регулярните изрази са символни низове със

специална структура, която отразява сложен критерии за търсене. По темата за регулярни изрази може да се напише цяла глава (и дори книга) и те се срещат в много други програмни езици, приложения и системи, поради което ние няма да ги разгледаме в настоящото изложение, но ще си позволим един разширен пример¹¹:

```
1 public class Main {
2     public static void main(String[] args) {
3         String stack = "The rain in Spain falls";
4         String[] parts = stack.split("in");
5         System.out.println(parts.length); //
6         System.out.println(parts[0]);    // "The ra"
7         System.out.println(parts[1]);    // " ";
8         System.out.println(parts[2]);    // " Spa"
9         System.out.println(parts[3]);    // " falls"
10
11         stack = "1;Ivan Petrov;Georgiev";
12         parts = stack.split("[;, ]");
13         // [1, Ivan, Petrov, Georgiev]
14         System.out.println(Arrays.toString(parts));
15
16         stack = "1 ; Ivan;;;Petrov   Georgiev";
17         parts = stack.split("[; ]+");
18         // [1, Ivan, Petrov, Georgiev]
19         System.out.println(Arrays.toString(parts));
20     }
21 }
```

Разпечатка 11. Демонстрация на метода **split**

На редове 4-9 е демонстрирано разделянето на символен низ с метода **split**. Подаденият на ред 4 параметър означава, че **split** ще „започва“ нова част всеки път, когато срещне последователността от символи “in”. При това “in” ще се интерпретира като разделител, т.е. няма да се добавя нито към предишната, нито към новата част. Както е видно от редове 6-9, отделните части се връщат под формата на масив от символни низове, който в нашия случай има 4 елемента.

Това, че подаденият параметър се интерпретира като регулярен израз означава, че някои символи имат специален смисъл. Така например определен брой символи, заградени в квадратни скоби, означават, че за дадената позиция в разделителя има различни варианти. Регулярният израз на ред 17 означава, че

11 Вж. например Friedl, J. Mastering Regular Expressions, 3rd Edition, O'Reilly Publishing, 2006

разделителят е една позиция, но за тази позиция има три варианта – интервал, запетайка или точка и запетая.

По същия начин регулярния израз на ред 17 обозначава, че разделителят е един или повече символа състоящи се от запетая, интервал или точка и запетая. Знакът плюс след символ или специална конструкция, обозначава че символът или конструкцията трябва да се повтарят един или повече пъти за да имаме съвпадение.

2.9. Метод `replace`

Методът **`replace`** създава нов символен низ от стария, като в процеса позволява заменяне на определена последователност от символи с друга. Методът има два варианта, като единият позволява замяна на единичен символ с друг такъв, а другият позволява подаването на цял подниз съответно с друг такъв.

<pre>String replace(char oldChar, char newChar) String replace(CharSequence Target, CharSequence Replacement)</pre>

Разглеждайки дефинициите на двете функции, наблюдателните читатели без съмнение са забелязали, че втората функция взема аргументи от тип **`CharSequence`** вместо аргументи от тип **`String`**. **`CharSequence`** е интерфейс, който се реализира от класа `String`, както и от класа **`StringBuilder`**, който ще бъде разгледан по-късно. Тъй като **`String`** реализира **`CharSequence`**, втората функция може да приема аргументи от тип **`String`**.

Без значение кой от двата метода се използва, първият аргумент е поднизът, който ще бъде заместван, докато вторият аргумент е поднизът, който ще бъде поставен на негово място в оригиналния низ.

Долният пример демонстрира използването и на двата метода.

<pre>1 public class Main { 2 public static void main(String[] args) { 3 String origin = "barbaron"; 4 origin.replace('b', 'g'); 5 System.out.println(origin); // barbaron</pre>

```

6      origin = origin.replace("bar", "gar");
7      System.out.println(origin); // gargaron
8  }
9  }

```

Разпечатка 12. Използване на методите **replace**

Редове 4-5 демонстрират, че заместването на текста не се извършва в оригиналния символен низ. Ред 4 извършва такова заместване, като замества всички срещания на буквата *b* с буквата *g*. Програмистът обаче не взема мерки да съхрани връщаната от **replace** референция към нов низ. Вместо това на ред 5 той отпечатва оригиналния низ.

Редове 6-7 представляват почти същата операция, но този път програмистът присвоява резултата от **replace** отново на променливата **origin**. Така референцията към стария низ се губи, а **origin** започва да сочи към новия, низ произведен от заместването.

Много е важно да се забележи, че **replace** замества всички срещания на низа, който ще се заменя. Така на ред 6 “bar” се замества с “gar” два пъти.

2.10. Метод contains

Методът **contains** е дефиниран по следния начин:

```

boolean contains(CharSequence s)

```

Той връща **true**, ако даден низ съдържа указания параметър подниз и **false** в противен случай, както е демонстрирано по-долу.

```

1  public class Main {
2      public static void main(String[] args) {
3          String origin = "barbaron";
4          System.out.println(origin.contains("aro")); // true
5          System.out.println(origin.contains("Bar")); // false
6      }
7  }

```

Разпечатка 13. Използване на метода **contains**

Разбира се, както е илюстрирано и от програмния код на разпечатката, функцията различава малки от големи букви.

3. Представяне на символните низове в Java

За разлика от другите типове, разгледани до момента, класът **String** не е елементарен тип. Това означава, че в променливите от тип **String** се съхранява само препратка (референция) към истинския символен низ в паметта на компютъра¹². Това способства бързодействието и ефективното управление на паметта. Това е така, тъй като в Java символните низове могат да бъдат много големи и копирането на стойности, например при присвояване на една променлива на друга, може да се окаже много бавно и да генерира излишно много заета памет. Работата с препратки обаче е и източник на потенциални затруднения, тъй като е възможно две (или дори повече) променливи да сочат към един и същ символен низ в паметта. Това крие опасност - промяната на стойността чрез едната променлива би означавало автоматично променяне на стойността, достъпна през другата променлива.

Като компромис между удобството на използване срещу бързодействието, в Java променливите от тип **String** са референции, но самите данни, към които тези референции сочат, са *непроменими*, т.е. не могат да се променят по време на изпълнение на програмата.

Това има две много важни последствия:

На първо място всяка операция, която на пръв поглед променя съдържанието на String променлива, всъщност създава копие на символния низ. Така се избягва ситуацията, при която две променливи сочат към една и съща стойност в паметта и манипулацията на една от тях променя стойността, сочена от другата. Долната програма илюстрира тази важна особеност.

1	<code>public class Main {</code>
---	----------------------------------

¹² За по-запознатите читатели можем да прецизираме като уточним, че променливите от елементарен тип се съхраняват изцяло в стека на програмата, докато при променливите от референтен тип данните се съхраняват в областта за данни /хийпа/ на програмата, а в стека се поставят само референции към тях. Нещо повече, константите от тип символен низ се съхраняват в специална област на паметта - т.нар. „таблица на символните низове“.

2	public static void main(String[] args) {
3	String str = "Infographics";
4	String a = "Infographics";
5	String b = a;
6	System.out.println(str==a); //true
7	System.out.println(str==b); //true
8	a = a+"6";
9	System.out.println(str==a); //false
10	System.out.println(b==str); //true
11	b = b+"6";
12	System.out.println(a==b); //false
13	System.out.println(a.equals(b)); //true
14	}
15	}

Разпечатка 14. Използване на метода **contains**

Въпреки че променливите **a** и **b** на пръв поглед са инициализирани отделно, Java се опитва да икономиса мястото, заето от константите – символни низове. Поради това по време на компилация компилаторът търси еднакви константи от символен тип. За всяка група еднакви константи, компилаторът поставя само един запис в символната таблица на програмата. Така променливите **a** и **b** всъщност биват инициализирани с една и съща референция (редове 4 и 5), която сочи към единственото копие на символния низ *“Infographics”* в паметта. При инициализирането на променливата **b**, същата референция, идваща от променливата **a**, се поставя и в **b**. По този начин към ред 6 имаме три променливи, които сочат към едно копие на константния символен низ *“Infographics”*.

На ред 8 към променливата **a** се добавя символния низ „6“. Това, което всъщност се случва не е промяна на оригиналния символен низ. Вместо това Java създава нов символен низ и поставя в него текста *“Infographics6”*. Тъй като резултатът от операцията се присвоява пак на **a**, променливата **a** сега сочи към *“Infographics6”*, и тя е различна от референциите, които сочат към стария низ и още се намират в **str** и **b**.

На ред 11 към **b** също се прибавя символния низ “6”. Сега в **a** и **b** имаме еднакви стойности. По време на изпълнение обаче Java не сверява символните стойности за еднаквост (това би отнело ценно време при изпълнение на програмата), така че всъщност слепването на низове на ред 6 създава нов низ

“Infographics6” в паметта. Така в променливите **a** и **b** има различни референции, сочещи към два различни символни низа “Infographics6” в паметта. Ето защо на ред 12 операторът за проверка за равенство връща **false**. На ред 13 е показан *правилния* начин за сравняване на текстови низове, който сравнява не референциите, а съдържанието на сочените низове.

Ако в дадена програма често се налага да се променя даден символен низ, това ще генерира голям брой копия на различни вариации на този символен низ. Разбира се, ако в даден момент някой от тези варианти повече не е сочен от никоя променлива, той е кандидат за освобождаване при следващата операция по „събиране на боклука“. За съжаление обаче операциите по събиране на боклука забавят програмата и не се случват мигновено, поради което е добра идея да не се създават излишно много копия на символните низове.

Долната програма илюстрира ситуация, при която копирането на символни низове създава много „боклук“ за събиране.

```
1 public class Main {  
2     public static void main(String[] args) {  
3         String base64alphabet = "" ;  
4         for (char a='A'; a<='Z'; a++) {  
5             base64alphabet = base64alphabet+a;  
6         }  
7         for (char a='a'; a<='z'; a++) {  
8             base64alphabet = base64alphabet+a;  
9         }  
10        for (char a = '0'; a<='9'; a++) {  
11            base64alphabet = base64alphabet+a;  
12        }  
13        base64alphabet = base64alphabet+"+/";  
14    }  
15 }
```

Разпечатка 15. Използването на **String** в цикли, които променят съдържанието на низа е лоша практика.

Показаният на разпечатката код всъщност може да бъде директна част от реална програма, която извършва Base64 кодиране. При това кодиране, произволни числени данни, състоящи се от байтове от по 8 бита, се разделя на последователности от по 6 бита, след което всяка от тези 6-битови последователности се представя с различна буква от азбука от 64 елемента,

основно букви и цифри. Азбуката е избрана така, че да се пренася ефективно между различни софтуерни програми и системи и всъщност се използва широко в електронната поща и уеб.

Програмата на разпечатка 15 демонстрира само съставянето на азбуката. Първият цикъл започва от празен низ, към който буква по буква се добавят всички големи букви от латинската азбука. Така този процес ще създаде символните низове „A”, “AB”, “ABC” и т.н. общо 27 низа. Всеки от тези низове ще се използва за кратко, докато на негова база се генерира следващия. Всички те веднага след използването си вече не се сочат от никоя променлива и подлежат на изчистване при събирането на боклука. По подобен начин останалите цикли ще добавят още елементи към азбуката и накрая ще имаме около 64 излишни символни низа, които да се изчистят. Това е само началото на алгоритъма по кодиране в Base64, който предвижда всеки три байта данни да се представят с 4 букви. Използването на **String** в хода на този процес ще предизвика създаването на нов String на всяко кодиране на следващите три байта. Ако говорим например за прикачване на файл 20 мегабайта към пощенско съобщение, това ще означава непосилно много копия от по 20 мегабайта в паметта на компютъра.

За да се избегне създаването на множество копия, демонстрирано по-горе, Java има специален клас, наречен **StringBuilder**, който се използва когато се нуждаем от използване на цикли за съставяне на низове. За съжаление подробното описание на функционирането на този клас е доста обемисто и няма да бъде обяснено в настоящето изложение.

*Второто много важно следствие от това, че символните низове са референции към реалните данни е това, че **те не могат да бъдат сравнявани по стандартния начин с оператор за сравнение**.* Операторът за сравнение е позволен, но той сравнява стойността на самата референция. Това означава, че за променливи от тип **String**, които сочат към един и същ символен низ в паметта операторът вярно ще връща стойност **true**. Но ако сравняваме референции към

два еднакви символни низа, разположени на различни места в паметта, той грешно ще връща **false**.

Правилният начин за сравняване на два низа е чрез метода **equals**, който е дефиниран по следния начин:

```
boolean equals(Object o)
```

Методът сравнява символните низове по съдържание и връща **true**, ако двата низа са еднакви. Читателите следва да забележат, че параметърът на **equals** е от тип **Object**. Това не трябва да ги смущава, тъй като този метод е наследен от клас на име **Object**. Ефектите на наследяването ще бъдат разгледани в следващите глави, но на този етап е добра новина, че това не влияе на нормалната работа по никакъв начин. Ако погрешка програмистът подаде за параметър референция към обект, който не е символен низ, резултатът от функцията ще е **false**.

Ако е необходимо два низа да бъдат подредени по азбучен ред, може да се използват методите **compareTo** и **compareToIgnoreCase**, дефинирани по следния начин:

```
int compareTo(String anotherString)  
int compareToIgnoreCase(String anotherString)
```

Методите сравняват текущия символен низ (т.е. този, чрез чиято променлива се извиква метода) с друг низ подаден като параметър (**anotherString**). Ако текущия символен низ е преди **anotherString** в азбучния ред, т.е. двата низа са в правилния ред, методите връщат положителна стойност. Ако обратното – текущият символен низ е след **anotherString** в азбучния ред, методите връщат отрицателна стойност. Ако двата символни низа са равни, методите връщат нула.

Разликата между двата метода е това, че първият прави разлика между малки и големи букви, докато вторият сравнява низовете, игнорирайки това дали буквите са големи или малки.

Използването на тези два метода е демонстрирано в последния за тази глава пример:

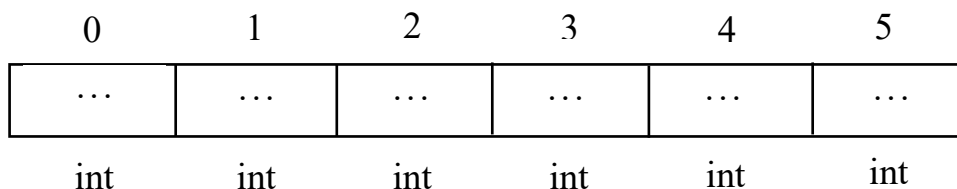
```
1 public class Main {  
2     public static void main(String[] args) {  
3         // отр. стойност - abc е преди abd по азбучен ред  
4         System.out.println("abc".compareTo("abd"));  
5  
6         // положителна стойност, в азбучния ред на компютъра  
7         System.out.println("bac".compareTo("BAC"));  
8  
9         // главните букви са преди малките и следователно  
10        // abc е след BAC.  
11        // 0 - низовете са равни  
12        System.out.println("abc".compareTo("abc"));  
13    }  
14 }
```

Разпечатка 16. Използване на **compareTo** и **compareToIgnoreCase**

Глава 6. Масиви

1. Понятие за масив

Масивите представляват множество еднотипни клетки в паметта, които се достъпват чрез една променлива. Всяка такава клетка се нарича „елемент на масива“. Елементите на масива са номерирани, така че всеки елемент има уникален числов номер, който на професионален език се нарича „индекс“. Индексите започват от нула, тоест първият елемент на масива има индекс 0, вторият има индекс 1 и т.н. (вж. фиг. 1)



Фигура 1. Схематично представяне на масив от 6 елемента от тип `int` в паметта на компютъра

2. Деклариране и инициализиране на масиви

За да ползваме масиви в Java, ние се нуждаем от променливи, които да сочат към тях в паметта на компютъра. **Самите масиви обаче се създават отделно от променливите.**

```
1 package uk.co.lalev.javabook;
2 public class Main {
3     public static void main(String[] args) {
4         byte[] age;           //деклариране на променлива
5         double[] totalPay;    //деклариране на променлива
6         short length[];      //деклариране на променлива. Лош стил!
7
8         age = new byte[100];   //създаване на масив
9         totalPay = new double[15]; //създаване на масив
10        length = new short[200]; //създаване на масив
11    }
12 }
```

Както е видно от редове 4-5 на разпечатката, декларацията на променливи от тип масив се състои от типа на масива и квадратни скоби, следвани от името на променливата. Декларацията има и валиден алтернативен вариант (показан на ред 6), при който квадратните скоби се поставят след името на променливата. Този вариант обаче не е широко приет сред програмистите на Java и се възприема като лош стил.

Масивите са референтни типове данни. Наред с други особености, това означава, че програмистът трябва сам да инициира заделянето на място в оперативната памет на компютъра. Ние наричаме този процес „**създаване на масива**“.

Създаването на масив става чрез извикването на ключовата дума **new**. След нея се изписват типът и броят на елементите в масива, като броят е заграден в квадратни скоби. Създаване на масив и присвояването му на променлива е демонстрирано на редове 8-10 на разпечатка 1.

Така например **new double[15]** на ред 9 на разпечатката създава масив с 15 елемента от тип **double**. Цялата декларация на реда създава масив и го присвоява на променливата **totalPay**, чийто тип е „масив от тип **double**“.

Процесът на деклариране на променливата и създаването на масива може да се извърши и наведнъж, както е демонстрирано в следващия програмен сегмент:

10	int[] ids = new int [60];
11	float[] coefficients = new float [20];

Разпечатка 2. Деклариране на променливи и създаване на масиви

След като масивът е вече създаден и присвоен на променлива, променливата може да бъде използвана за достъп до елементите на масива. Достъпът до конкретен елемент се указва, като след името на променливата в квадратни скоби се постави неговия индекс (вж. разп. 3).


```

1 package uk.co.lalev.javabook;
2
3 public class ArrayDemo {
4
5     public static void main(String[] args) {
6         int[] primeNumbers = new int[10];
7         primeNumbers[0]=2;
8         primeNumbers[1]=3;
9         primeNumbers[2]=5;
10        primeNumbers[3]=7;
11        primeNumbers[4]=11;
12
13        System.out.println(primeNumbers[0]); // Извежда 2
14        System.out.println(primeNumbers[7]); // !!! Извежда 0
15        System.out.println(primeNumbers[10]); // Извежда грешка по време
16                                                // на изпълнение на програмата
17    }
18 }

```

Разпечатка 3. Достъп до елементите на масив

Така например декларацията на ред 7 съдържа оператор за присвояване, който задава стойност 2 на този елемент на масива **primeNumbers**, който има индекс 0. С други думи задаваме стойност 2 на нулевия елемент на масива.

За да избегнем твърде „натоварени“ и сложни изречения при обясненията, ние ще наричаме масивите с имената на променливите, които сочат към тях. Читателите обаче трябва винаги да мислят за масивите като отделни същности. Както ще бъде демонстрирано по-нататък, обратното води по много начини до неочаквани и опасни грешки.

При създаване на масиви Java поставя подходяща стойност по подразбиране във всеки техен елемент. За елементарните числови типове това е числото 0, докато за **boolean**, това е стойността **false**.

Това е причината ред 14 на програмата от разпечатка 3 да отпечатва нула. За разлика от елементите от 0 до 4, елементът с индекс 7 в масива **primeNumbers** никога не е получавал стойност от програмиста. При създаването на масива обаче, всички елементи са получили начална стойност нула.

Ред 15 на разпечатката използва възможността да илюстрира често допускана грешка при работа с масиви. Масивът **primeNumbers** е от десет елемента с индекси от 0 до 9. Ето защо при опит да се достъпи елемент с индекс

10 резултатът е грешка, която се проявява чак при изпълнение на програмата. Програмистите на Java трябва да са извънредно внимателни и да не „излизат“ извън валидните за масива индекси при извършване на операции с неговите елементи.

Инициализирането на елементите на масивите, тоест поставянето в тях на начални стойности, различни от нула, е типична част от много алгоритми. В същото време то е неудобно за извършване елемент по елемент. За щастие има удобен начин за инициализиране на всички елементи, който е илюстриран от програмния сегмент на разпечатка 4.

9	<code>int[] test = new int[] {1, 3, 5}; //Излишно сложно!!!</code>
10	<code>int[] primeNumbers = {2,3,5,7,11,13,17,19};</code>
11	<code>double[] squareRoots = {1.41, 1.73, 2.24, 2.65};</code>

Разпечатка 4. Достъп до елементите на масив

Ред 9 демонстрира, че за да се попълни масива с начални стойности е достатъчно техният списък да се постави във фигурни скоби след ключовата дума *new* и типа на масива. В този случай декларацията много прилича на досега разгледаните, с тази разлика, че в квадратните скоби след ключовата дума *new* и типа на масива не е посочен броят на елементите на бъдещия масив. Компиляторът на Java ще създаде новия масив с толкова елементи, колкото стойности има във фигурните скоби.

Така оформеният запис е доста дълъг и съдържа елементи, които се повтарят. Когато са подадени стойности за инициализация, компилаторът на Java може сам да определи типа на бъдещия масив според тези стойности. Ето защо синтаксисът за създаване на инициализиран масив отдавна е опростен, като от него са изпуснати ключовата дума **new** и типа на масива.

Опростеният запис е демонстриран на редове 10 и 11. Именно този начин за създаване на инициализирани масиви се счита за добър стил на програмиране.

3. Многомерни масиви

Разгледаните досега масиви са подходящи за представяне на числа, които могат да се разглеждат като последователност. Тези масиви са известни на професионален език като „**едномерни**“.

Доста често се налага в паметта на компютъра да се съхраняват таблици или дори „кубове“ и многомерни структури от еднотипни елементи. За такива ситуации Java предлага средства за реализиране на **многомерни** масиви.

Разпечатка 5 илюстрира използването на двумерни и тримерни масиви в Java. Както е видно от разпечатката, работата с тези масиви на синтактично ниво изглежда почти еквивалентна на работата с едномерни масиви. На мястото на един индекс при достъпа до клетка на масива, сега се използват два или повече индекса в зависимост от размерността на масива. По същия начин, при създаването на масив се посочват два и повече размера, които описват големината във всяко измерение.

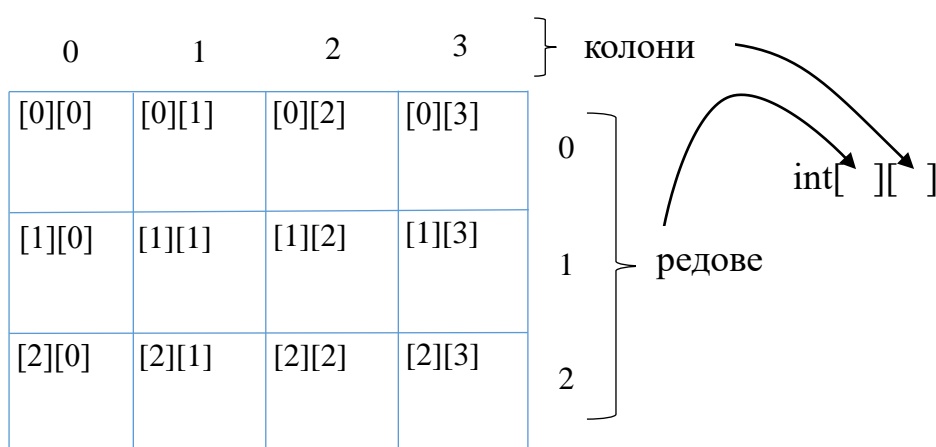
```
1 package uk.co.lalev.javabook;  
2  
3 public class MultidimensionalArrays {  
4  
5     public static void main(String[] args) {  
6         int[][] matrix = new int[5][10];  
7         double[][][] coefficients = new double[5][10][10];  
8  
9         matrix[0][1]=5;  
10        matrix[0][9]=3;  
11        matrix[1][3]=4;  
12        matrix[1][4]=matrix[1][2]+matrix[0][1];  
13        coefficients[1][2][0]=3.14*matrix[1][4];  
14    }  
15 }
```

Разпечатка 5. Реализация на многомерни масиви в Java

Докато използването на едномерни масиви е интуитивно, използването на двумерни и многомерни масиви от страна на начинаещите програмисти първоначално се сблъсква с проблеми. В известен смисъл геометричната интуиция пречи на възприемане на по-абстрактно разбиране на двумерните и тримерните масиви, което помага значително при работата с тях. За да

илюстрираме проблема, ще зададем въпрос на читателите - „При наличие на таблица от данни кой от двата индекса на съответния масив трябва да обозначава редовете и кой – колоните?“.

Отговорът е, че всъщност няма значение. Програмистът може да избере да постави номерата на редовете в първия индекс и номерата на колоните във втория. При този избор таблица с цели числа от 10 реда и 5 колони се помещава в масив `int[10][5]`. При обратния избор нужните размери на масива са `int[5][10]`. По традиция, заета от математиката обаче, повечето програмисти биха избрали да поставят редовете в първия индекс на масива (вж. фиг. 2).



Фигура 2. Схематично представяне двумерен масив от три реда и четири колони

Java всъщност третира масивите напълно абстрактно и ги разглежда като масиви от масиви. Така например типът `int[][]` се интерпретира като “масив от тип масив от `int`”. Въпросите за начините за съпоставка на реалните данни (вж. фиг 6.2) с тези структури всъщност опира до решението на програмиста, и е проблем, който е напълно изнесен извън обхвата на езика. При определянето на индексите е полезно да мислим именно по този абстрактен начин, който по-добре отговаря на начина по който Java борави с масивите.

На този етап читателите могат да използват рапечатка 6 за да проверят своята интуиция и разбиране за индексите и размерите на масивите като открият редовете, които ще произведат грешка при изпълнение на програмата!

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //!!! Тест. Някои редове могат да съдържат немаркирани грешки!
7
8         int a[][]=new int[3][7];
9         int b[][][]=new int[1][2][3];
10
11         a[1][2]=b[2][3][1];
12         b[1][1][1]=a[3][7];
13         b[1]=a;
14         a[1]=b[1][2];
15     }
16 }

```

Разпечатка 6. Реализация на многомерни масиви в Java

Всички редове между 11 и 14 включително ще произведат грешки при компилация. Причината е, че при всички тях първият индекс на масива *b* е невалиден. Масивът е дефиниран като `int[1][2][3]`. Тоест това е масив с един елемент от тип `int[2][3]` и следователно единственият валиден първи индекс на **b** може да бъде 0. По същия начин вторият индекс на **b** може да бъде 0 или 1, а третият не може да навишава 2.

В допълнение, на ред 12 има допълнителна грешка - `a[3][7]` няма как да сочи към валиден елемент от масива, тъй като самият масив е дефиниран като `int[3][7]`. Тоест максимумът за първият индекс е 2, а за вторият – 6 и „максималният“ елемент тогава е `int[2][6]`.

Ако се абстрахираме от грешките, редове 13 и 14 са интересни и за друго. На ред 13 вместо три индекса на масива **b**, ние използваме един. Това е допустимо именно понеже Java разглежда масивите като „масиви от масиви“. Ако индексът на **b** бе 0, ред 13 всъщност щеше да е валидна декларация. Тъй като типът на елемента `b[0]` е `int[][]` и масивът **a** е точно от този тип, присвояването щеше да е валидно. Ред 14 присвоява на `a[1]` (елемент от тип `int[]`) `b[1][2]` (елемент, който също щеше да е от тип `int[]`, ако съществуваше). При условие, че `b[1][2]` съществуваше, присвояването щеше да е валидно.

Като последна важна демонстрация, разпечатка 7 показва как се извършва инициализацията на многомерен масив.

12	int [][] matrixB = {{1,2}, {3, 4}, {5, 6}};
13	System.out.println(matrixB[0][0]); //1
14	System.out.println(matrixB[0][1]); //2
15	System.out.println(matrixB[1][0]); //3
16	System.out.println(matrixB[1][1]); //4
17	System.out.println(matrixB[2][0]); //5
18	System.out.println(matrixB[2][1]); //6

Разпечатка 7. Инициализация на многомерни масиви в Java

matrixB е масив от тип масив, като всеки елемент от него е масив от тип **int[]**. Стойностите за инициализация са организирани по същия начин. Така стойностите {1,2} във вътрешните скоби са стойност за **matrixB[0]**. По същия начин {3, 4} и {5, 6} са стойности за **matrixB[1]** и **matrixB[2]**.

Завършваме с това как може да се получи информация за размера на даден масив. Тази операция е нужна например, когато масиви се предават като параметри от един метод на друг. Всяка променлива от тип масив има поле, което съдържа броя на елементите в масива. Това поле се нарича **length** и може да се достъпи успешно, при положение, че променливата на масива сочи към реален масив в хийпа на програмата. Долната програма демонстрира използването на **length** в няколко различни сценария.

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Програмата е стартирана с "+args.length+
7                             " аргумента!");
8
9         int [][][] p = new int[5][4][2];
10        System.out.println(p.length);           // 5
11        System.out.println(p[1].length);        // 4
12        System.out.println(p[0].length);        // 4
13        System.out.println(p[0][0].length);     // 2
14
15        int[][] q = {{1,2}, {3,4}, {5,6,7}};
16        System.out.println(q);                  // 3
17        System.out.println(q[0].length);        // 2
18        System.out.println(q[1].length);        // 2
19        System.out.println(q[2].length);        // 3
20
21        int[] d; //!!!
22        System.out.println(d.length); //Грешка при компилация!
23    }
24 }

```

Разпечатка 8. Използване на **length** за определяне на броя на елементите на масив

Ред 6 на програмата от разпечатката започва с определяне на броя на аргументите, подадени от командния ред при стартиране на програмата. Java предава тези аргументи именно като масив от тип **String[]** на метода **main**.

На ред 9 програмистът дефинира тримерен масив с размери 5,4 и 3 и го присвоява на променливата **p**. **p.length** показва размера на „най-външния“ масив. Всеки от елементите в този масив на свой ред е масив от тип **int[][]**. И тъй като са създадени заедно с декларацията на ред 6, тези масив-елементи имат еднакъв брой елементи – 4. Това е отпечатано на редове 11 и 12 за елементите **p[0]** и **p[1]**. На ред 13 по същия начин е отпечатана дължината на елемента **p[0][0]**, който на свой ред е масив от тип **int[]**.

На ред 15 се създават нов масив и нова променлива. Този път е използван инициализатор с необичайни свойства. Инициализаторът създава масив от елементи, които на свой ред са масиви от тип **int[]**. Тези елементи – масиви не са с еднаква дължина, както е демонстрирано на редове 17-19. Първите два са от два елемента, докато последният е с дължина 3 елемента. Декларацията на ред 15 е

валидна именно защото Java разглежда многомерните масиви като „масиви от масиви“.

На редове 21-22 обаче е демонстрирано невалидно използване на **length**. Променливата **d** не е инициализирана, т.е. няма масив, към който тя да сочи. Извикването на **d.length** при това условие предизвиква грешка при компилация.

6. Свойства на масивите

Масивите са реализирани като референтен тип от данни. С други думи, променливите от тип масив съдържат препратка към реалното местоположение на масива в хийпа на програмата. Долните разпечатки демонстрират малко по-особените свойства на масивите, произтичащи от този факт.

9	int[] ar;	// създаване на променлива от референтен тип в стека
10		
11	ar = new int [5];	// създаване на масив в хийпа на програмата
12		// и присвояването му на променлива a.
13	ar[0] = 3;	// изп. на променливата за достъп до елемент в масива
14		
15	int[] br;	// създаване на нова променлива от реф. тип в стека
16	br[0] = 3;	//Грешка!!!

Разпечатка 9. Променливи от референтен тип в Java

Както беше вече демонстрирано, декларирането на променливи от тип масив не е равносилно на създаване на масива. Това е вярно не само за масивите, но и за всички променливи от референтен тип.

В програмния сегмент 9 програмистът създава променлива от тип масив. Тази променлива има име **ar** и може да сочи към *всеки* масив от цели числа в хийпа на програмата *без значение колко са неговите елементи*. Тъй като променливата ще съдържа само препратка към масива в хийпа, нейният размер е малък и фиксиран, което означава, че се побира в стека на програмата без проблеми.

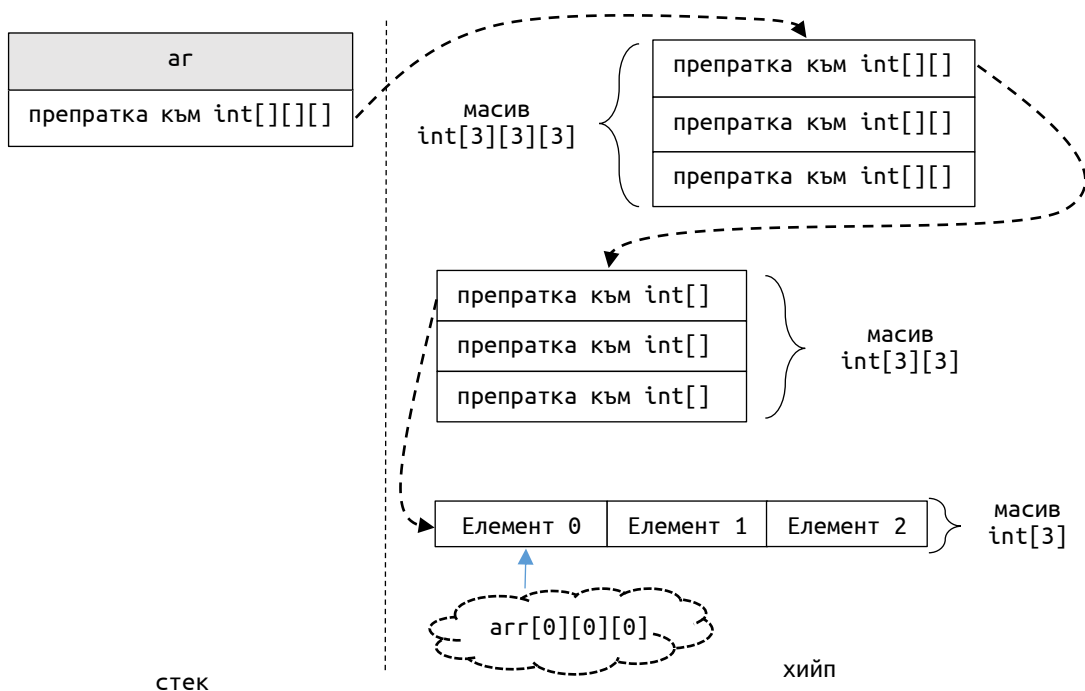
На ред 10 променливата е декларирана и създадена, но още *не сочи* към реален масив в хийпа на програмата. Ако на този етап програмистът се опита да

достъпи елемент, това ще генерира грешка. Едва на ред 11 променливата получава стойност, която всъщност е необходимата препратка към новосъздаден масив с 5 елемента от тип **int**.

Оттук нататък програмистът ще може да използва променливата **ar** за достъп до елементите на масива, стига подаденият индекс да отговаря на размерите на масива. Успешен достъп до масива **ar** е демонстриран на ред 13.

На ред 15 програмистът създава друга променлива от тип масив с име **br**. Тази променлива също първоначално не сочи към масив. Ред 16 демонстрира, че при това положение опитът да се достъпят елементи чрез нея логично води до грешка по време на изпълнение на програмата.

С направените уточнения относно работата на референтните типове, става възможно да се обясни по-прецизно какво имаме предвид „масив от масив“. Долната графика прави опит да илюстрира как изглеждат в паметта масив от тип **int[3][3][3]** и променливата **ar**, която сочи към него.



Фигура 3. Разположение на тримерен масив от цели числа в паметта на компютъра

Референтният характер на масивите води и до други неинтуитивни за начинаещите програмисти свойства. Следният сегмент от код илюстрира ефекта на оператора за присвояване върху променливи от референтен тип. В конкретния случай тези променливи са масиви:

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         int[] a = new int[5];
7         a[0]=12;
8
9         /* Сега масивът има стойности {12,0,0,0,0} */
10        System.out.println(a[0]); // 12
11
12        int[] b = a;
13        System.out.println(b[0]); // Извежда 12 !!!
14
15        b[1] = 3;
16        System.out.println(a[1]); // Извежда 3 !!!
17    }
18 }
```

Разпечатка 11. Множество променливи сочат към един масив

На редове 6-10 на разпечатка 11 програмистът извършва съвсем нормални операции с масива. Той създава променлива и масив от 5 елемента, като използва променливата, за да присвои стойността 12 на първия елемент от масива.

На пръв поглед странните резултати започват на ред 12. Там програмистът създава нова променлива от тип масив и за пръв път досега, той използва оператора за присвояване, за да присвои на тази променлива *друга такава*.

Когато програмистът използва тази нова променлива – **b** – първият елемент на масива има стойност 12. Това не е толкова изненадващо - все пак присвоихме **a** на **b** и е логично двете променливи да са „равни“ и “еднакви“. Както показват долните два реда обаче, когато променим елемент на **b**, ние променяме и съответния елемент на **a**. Това се дължи на факта, че двете променливи сочат към един и същи масив в хийпа на програмата!

Когато в Java присвояваме стойността на една променлива от референтен тип на друга променлива от същия тип, Java копира само

референцията и като резултат двете променливи сочат към една и съща структура (обект, масив или текстов низ) в хийпа на програмата. Това поведение е напълно различно от поведението на оператора за присвояване по отношение на променливите от елементарен тип. При присвояване на стойности между променливи от елементарен тип, Java копира *стойността* на едната елементарна променлива в другата. Оттам нататък тези променливи и техните стойности са независими една от друга.

Очертаното най-главно свойство на променливите от референтен тип е толкова важно, че заслужава да бъде илюстрирано още веднъж:

```
1 package uk.co.lalev.javabook;
2
3 public class ReferenceDemo {
4
5     public static void main(String[] args) {
6         int a = 5;
7         int b = a;
8         b = 7;
9         System.out.println(a); //5
10        System.out.println(b); //7
11
12        int[] ar = new int[1];
13        ar[0] = 5;
14
15        int[] br = ar;
16        br[0] = 7;
17
18        System.out.println(ar[0]); //7
19        System.out.println(br[0]); //7
20    }
21 }
```

Разпечатка 12. Множество променливи сочат към един масив

На ред 6 програмистът дефинира променлива **a** от елементарен тип. На ред 7 програмистът присвоява *стойността* на тази променлива на нова променлива с име **b**. Стойността на **b** се променя на ред 8. Тъй като **b** и **a** са напълно независими една от друга променливи оттук нататък те имат различни стойности. **a** има оригиналната си стойност – 5. **b** има новата си стойност – 7.

На ред 12 е дефинирана променлива с име *ar*. В рамките на същата декларация се създава масив от един елемент и референцията към него се поставя

в *ar*. На ред 13 чрез променливата *ar* се поставя стойност 5 в единствения елемент на масива.

На ред 15 се създава нова променлива от тип масив (по точно масив от цели числа) и име **br**. На същия ред на новата променлива се присвоява променливата **ar**. Операторът за присвояване извършва същата операция като при елементарните типове – стойността на променливата **ar** се поставя в променливата **br**. Тъй като обаче променливите са от референтен тип, техните конкретни стойности са препратките към масивите. Като резултат променливата **br** получава копие на препратката в променлива **ar**. Сега двете променливи сочат към един и същ масив в паметта на компютъра. Когато чрез едната променлива променим елемент на масива, това се вижда и при използване на другата променлива, тъй като това е един и същ масив!

След тези обяснения, най-накрая можем да демонстрираме някои „гранични“ случаи, които се появяват доста по-често на интервюта за работа, отколкото в реални ситуации!

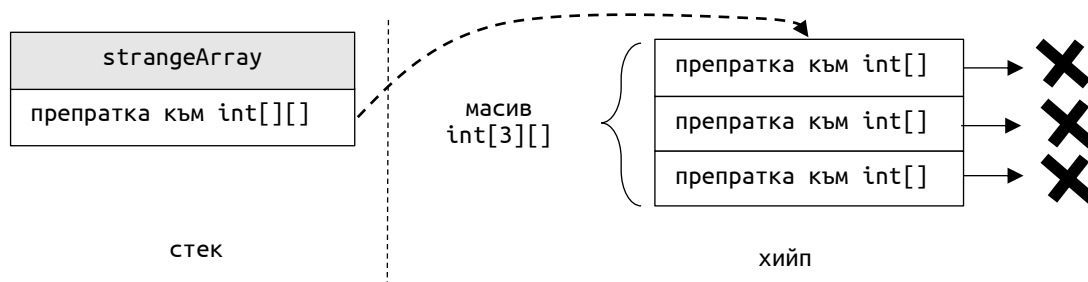
```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         int[][] strangeArray = new int[3][];
7         int[] zero = new int[10];
8         int[] one = new int[9];
9         int[] two = new int[8];
10
11         strangeArray[0]=zero;
12         strangeArray[1]=one;
13         strangeArray[2]=two;
14
15         System.out.println(strangeArray[0][9]);
16         System.out.println(strangeArray[1][9]); //Грешка при изпълнение
17     }
18 }
```

Разпечатка 13. Многомерните масиви в Java са масиви от референции към масиви

Най-странно изглеждащият ред на разпечатка 13 е ред номер 6. Там ключовата дума **new** е извикана с параметър масив от тип **int[3][]**. За разлика от

досегашните примери, при които винаги в скобите имаше конкретни числа, показващи размерите на масива, тук втората скоба е празна. Това е валидна декларация и чрез нея програмистът указва, че последното „ниво“ на структурите не трябва да се бъде създавано (вж. фиг. 4).

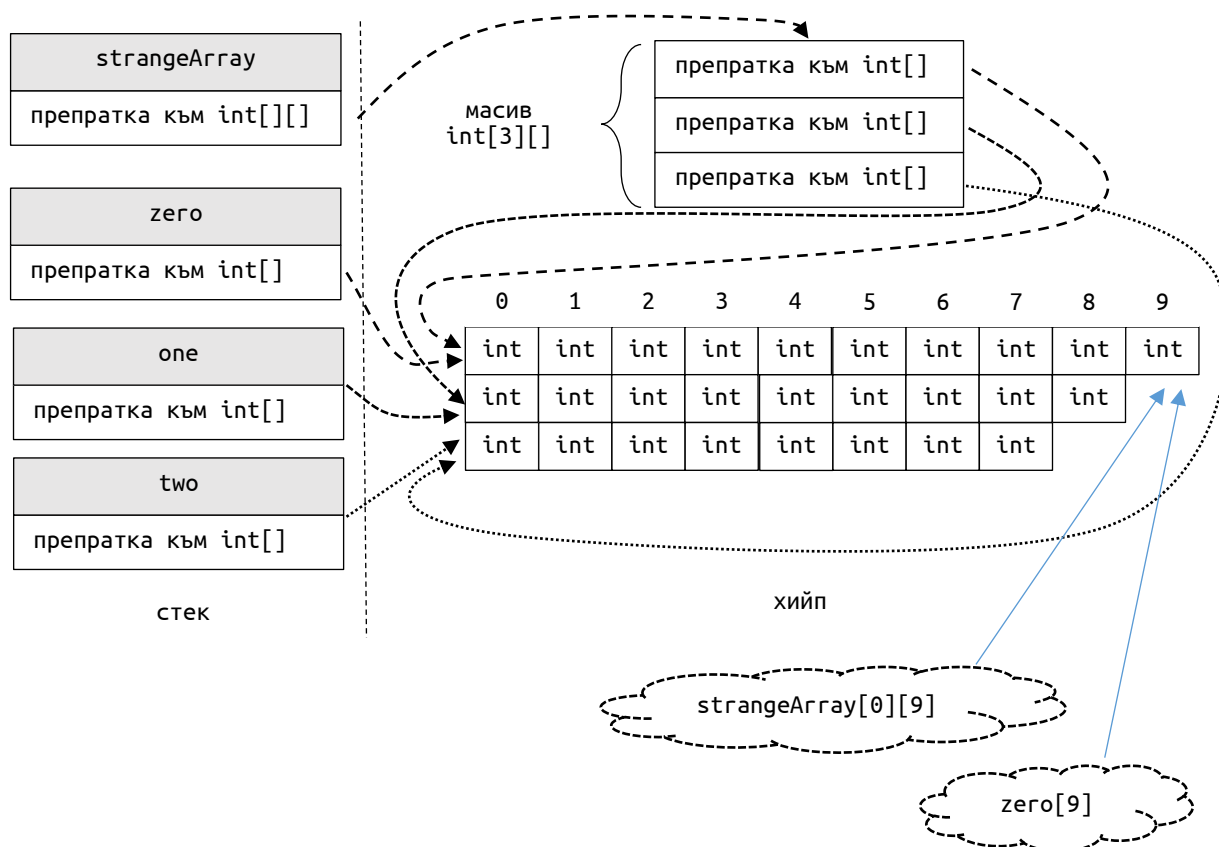
Ако непосредствено след изпълнението на декларацията на ред 6 на разпечатката програмистът се опита да достъпи елемента **strangeArray[0][0]**, това ще доведе до грешка. Причината е, че докато първото от двете нива (масив с препратки към масиви от тип **int[]**) е създадено, второто и последно ниво (масиви с елементи **int**) не е.



Фигура 4. Променливата **strangeArray** и съответстващият ѝ масив непосредствено след изпълнението на декларацията на ред 6 на разп. 13

Ако програмистът се опита да използва масива като едномерен, например като **strangeArray[0]**, това ще бъде напълно валидна декларация, тъй като първото ниво, споменато горе, е коректно. По този начин програмистът ще получи стойност от тип „препратка към масив от **int**“ или, написано в синтаксиса на Java – **int[]**.

Както е демонстрирано на редове 6-13 програмистът може да използва **strangeArray[0]** като едномерен и „ръчно“ да го зареди с препратки към валидни едномерни масиви. По този начин обаче е възможно всеки от едномерни тези масиви да бъде с различна дължина (вж. фиг. 5).



Фигура 5. Променливата **strangeArray** и съответстващият ѝ масив непосредствено след изпълнението на декларацията на ред 13 на разп. 13

Завършваме дискусията на темата за масивите с малко общи коментари относно логиката на описаното поведение на референтните типове и в частност – променливата от тип масив.

Може би **най-важното наблюдение**, което трябва да бъде направено е, че структурите, създадени в хийпа на програмата, *не* се управляват напълно автоматично от Java. Вместо това Java изчаква сигнал от програмиста кога да създаде структурата. Именно това е функцията на ключовата дума *new*. Тъй като обектите в хийпа са големи, това поведение има смисъл и позволява гъвкаво управление на паметта. Същото в пълна сила може да се каже за факта, че присвояването на една променлива от референтен тип на друга не води до автоматичното копиране на потенциално много голям обект от едно място на паметта на друго. Копирането още е възможно, но за целта програмистът трябва

да използва вградената библиотека с класове, както е показано в следващата точка.

5. Класът `java.lang.Arrays`

Класът `java.lang.Arrays` предлага редица методи, които улесняват работата с масиви.

Първият метод, който ще разгледаме, е методът `Arrays.toString()`. Този метод връща символен низ, който представя съдържанието на масива в удобен за четене и отпечатване вид, както е демонстрирано в долната програма:

```
1 package uk.co.lalev.javabook;
2 import java.util.Arrays;
3
4 public class Main {
5     public static void main(String[] args) {
6         int[] data = {1, 2, 3};
7         String[] names = {"Елена", "Виктор", "Елина"};
8         System.out.println(data); // [I@3fee733d
9         System.out.println(Arrays.toString(data)); // [1, 2, 3]
10        System.out.println(Arrays.toString(names)); // [Елена, Виктор, Елина]
11    }
12 }
```

Разпечатка 6. `Arrays.toString()` позволява отпечатването на масиви във удобен за четене вид

На ред 8 програмистът отпечатва директно променливата **data** от тип `int[]`. Както и при другите референтни типове това не води до отпечатване на стойностите на масива, като вместо това бива показана специална хеш сума. Тя не е особено полезно за извеждане или отстраняване на грешки в потребителските програми, тъй като програмистът в такъв случай по-скоро би искал да види съдържанието на масива. За щастие `Arrays.toString()` предлага именно тази функционалност. От съдържанието на масива се формира четлив символен низ, който може да се изведе със `System.out.println()`. Действието на този метод върху масиви от два различни типа е показано на редове 9-10 на разпечатката.

Следващият важен метод е **Arrays.fill()**, който позволява автоматичното запълване на масива с дадена стойност, различна от стойността по подразбиране. Както повечето методи, които ще разгледаме в настоящата точка, „методът“ **fill** всъщност е фамилия от предефинирани методи¹³, които позволяват функционалността по запълване да се извиква по еднакъв начин за различни типове от данни.

Начините за извикване на **fill** са два и са илюстрирани схематично в долния пример:

<code>fill(масив,стойност)</code>
<code>fill(масив, начален_индекс, краен_индекс, стойност)</code>

Първият вариант запълва целия масив със стойности, докато вторият начин позволява запълване на конкретна последователност от елементи, която не задължително обхваща целия масив.

```
1 package uk.co.lalev.javabook;
2
3 import java.util.Arrays;
4
5 public class Main {
6     public static void main(String[] args) {
7         int[] a = new int[5];
8         System.out.println(Arrays.toString(a)); // [0, 0, 0, 0, 0]
9         Arrays.fill(a, 4);
10        System.out.println(Arrays.toString(a)); // [4, 4, 4, 4, 4]
11        Arrays.fill(a, 2, 4, 2);
12        System.out.println(Arrays.toString(a)); // [4, 4, 2, 2, 4]
13    }
14 }
```

Разпечатка 14. Използване на **Arrays.fill()** за запълване на масиви със стойности

Програмата от разпечатката започва по същество на ред 7, на който програмистът създава масив от 5 елемента. Те се запълват със стойност по подразбиране за дадения тип данни, която в случая е нула. Това е видно от ред 8. Програмистът използва **Arrays.fill()** за да запълни масива със четворки (ред 10).

¹³ Ще разгледаме предефинираните методи по-нататък в изложението.

Последното извикване на метода на ред 11 запълва само елементите с индекс 2 и 3 с двойки. Читателите следва да забележат, че вторият и третият параметър на функцията на ред 11 задават диапазон от елементи. Долната граница се запълва, докато горната граница остана непроменена.

Фамилията предефинирани методи **Arrays.sort()** сортира елементите на масива във възходящ ред. Начините за извикване на методите са основно три, като най-гъвкавият от тях изисква използването на *Comparator*, поради което ще бъде разгледан по-нататък. Всички методи работят директно върху масива и не връщат стойност. Очертаните по долу начини позволяват съответно сортиране на целия масив или на част от него, затворена между два индекса.

<code>sort(масив)</code>
<code>sort(масив, начален_индекс, краен_индекс)</code>

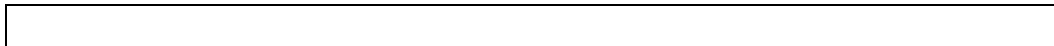
Конкретното използване на методите е демонстрирано в долната разпечатка:

```
1 package uk.co.lalev.inputoutput;
2
3 import java.util.Arrays;
4
5 public class Main {
6     public static void main(String[] args) {
7         int[] arr1 = {5, 2, 4, 3, 1};
8         Arrays.sort(arr1);
9         System.out.println(Arrays.toString(arr1)); // [1, 2, 3, 4, 5]
10        char[] arr2 = {'b', 'c', 'a', 'z', 'f'};
11        Arrays.sort(arr2, 0, 3);
12        System.out.println(Arrays.toString(arr2)); // [a, b, c, z, f]
13    }
14 }
```

Разпечатка 6. Използване на **Arrays.sort()** за сортиране на масиви

Arrays.binarySearch() позволява търсенето на индекса на дадена стойност, т.е. позицията, която стойността заема в масива. **Arrays.binarySearch()** работи само върху възходящо сортирани масиви. Общата форма на извикване на масива наподобява предните методи:

<code>binarySearch(масив, стойност)</code>
<code>binarySearch(масив, начален_индекс, краен_индекс, стойност)</code>



Ако търсената стойност фигурира в масива, **Arrays.binarySearch()** връща като резултатна стойност индекса на тази стойност. Ако търсената стойност липсва, връщаната стойност е отрицателна и указва мястото в масива, където търсената стойност може да бъде вмъкната, за да се запазят елементите в сортиран вид. Тъй като валидните индекси започват от 0, връщаната стойност е позицията с отрицателен знак, намалена допълнително с единица. Така връщана стойност -1 означава, че елементът трябва да се вмъкне на позиция 0, за да се запази сортирания вид на масива. По подобен начин -2 означава вмъкване на позиция 1 и т.н.

Ако даденият сортиран масив съдържа повече от един елемент с търсената стойност, методът ще върне индекса на един от тези елементи, като няма гаранция, че това ще е индексът на първият от тях.

```
1 package uk.co.lalev.inputoutput;
2
3 import java.util.Arrays;
4
5 public class Main {
6     public static void main(String[] args) {
7         int[] arr1 = {2, 5, 7, 7, 7, 11, 13};
8         System.out.println(Arrays.binarySearch(arr1, 5)); // 1
9         System.out.println(Arrays.binarySearch(arr1, 3)); // -2
10        System.out.println(Arrays.binarySearch(arr1, 0)); // -1
11        System.out.println(Arrays.binarySearch(arr1, 7)); // 3
12        if (Arrays.binarySearch(arr1, 6)<0) {
13            System.out.println("В масива няма стойност 6!");
14        }
15    }
16 }
```

Разпечатка 17. Използване на **Arrays.binarySearch()**

На ред 8 от разпечатката, програмистът търси стойност 5 в масива *arr1*. Такава стойност има и масивът е възходящо сортиран, така че **Arrays.binarySearch()** ще върне 1 – индексът на елемента със стойност 5 в масива. Ред 9 демонстрира ситуация, в която търсената стойност липсва. Числото 3 трябва да бъде вмъкнато на позиция 1, за да се запази сортировката, поради което резултатът от извикването на метода е -2.

На ред 11 се търси стойност 7. Първата седмица в масива е на позиция 2, но **Arrays.binarySearch()** връща 3, което демонстрира, че когато в търсената стойност се повтаря в масива множество пъти, методът може да върне индекса на който и да е от елементите с търсената стойност.

Последните редове илюстрират, че когато не се интересуваме от вмъкването на стойността, а само от това дали тя е в масива (това е типичният случай) е достатъчно да проверим, че резултатът от метода е неотрицателен.

Много често в практиката на програмистите се налага да бъдат сравнявани два масива. За целта **Arrays** предлага фамилия от методи **equals()** и **deepEquals()**.

Долният програмен сегмент демонстрира някои особености на използването на тези методи.

```
7  int[] arr1 = {2, 5, 7, 7};
8  int[] arr2 = new int[4];
9  arr2[0]=2;
10 arr2[1]=5;
11 arr2[2]=7;
12 arr2[3]=7;
13 System.out.println(arr1==arr2);           //false
14 System.out.println(Arrays.equals(arr1,arr2)); //true
15
16 int[][] arr3 = {{1,2}, {3,4}};
17 int[][] arr4 = new int[2][2];
18 arr4[0][0]=1;
19 arr4[0][1]=2;
20 arr4[1][0]=3;
21 arr4[1][1]=4;
22 System.out.println(arr3==arr4);           //false
23 System.out.println(Arrays.equals(arr3,arr4)); //false
24 System.out.println(Arrays.deepEquals(arr3,arr4)); //true
```

Разпечатка 18. Използване на **Arrays.equals()** и **Arrays.deepEquals()**

На редове 7-12 в разпечатката за целите на демонстрацията се създават два отделни масива, които обаче съдържат еднакви елементи. Масивът **arr1** на ред 7 е инициализиран от създаването си с елементите 2,5,7 и 7. Масивът **arr2** обаче съдържа същите елементи.

Когато двете променливи се сравнят със оператора за равенство (ред 13) резултатът е **false**. Това е така, тъй като операторът за равенство при референтните типове сравнява само референциите. **arr1** и **arr2** са референции към два различни масива в хийпа на програмата, поради което те не са равни. Ако

искаме да сравним самите елементи, ние бихме написали цикъл, който обхожда всички елементи индекс по индекс и ги сравнява. За щастие тази функционалност е вече налична в метода **Arrays.equals()**. Този метод ще сравни не референциите, а самите елементи. Това сравнение за **arr1** и **arr2** ще даде **true**, тъй като макар и различни масиви, на ред 14 в програмата те съдържат еднакви стойности елемент по елемент.

Методът **Arrays.equals()** е предвиден за едномерни масиви. Методът, който може да работи върху многомерни масиви е **Arrays.deepEquals()**. Редове 16-24 демонстрират някои особености на втория метод.

На редове 16-21 целенасочено създаваме два различни двумерни масива, които съдържат едни и същи елементи. Сравняването на референциите на ред 22 очаквано дава **false** точно като предната илюстрация на ред 13. Изненадващо обаче, на ред 23 методът **equals()** също връща **false**. Това най-общо се дължи на това, че методът **equals()** сравнява отделните елементи по тяхната стойност. Когато тази стойност на свой ред е референция (както е при двумерните и многомерните масиви), това дава грешни резултати. **deepEquals()** компенсира това и при срещане на референция към масив, сравнява рекурсивно този масив елемент по елемент. Това произвежда верния резултат в нашия пример на ред 24. Програмистите винаги следва да използват **deepEquals()** за сравнение на многомерни масиви.

Глава 7. Методи

1. Въведение в методите

Започваме въвеждането в класовете, обектите и методите по необичаен начин с малка програма. Тази програма използва повечето от механизмите на езика Java, които разглеждахме до момента. Тя има променливи, масиви, цикли и условни конструкции, но всъщност не използва нито методи, нито класове, нито обекти (поне не и по начина, по който те се използват в Java).

Програмата претърсва няколко масива с лични данни на клиенти по зададен клиентски номер и отпечатва имената на клиента, който има този номер:

[illegible]

```

22         for (int i=0; i<clientNumbers.length; i++) {
23             if(clientNumbers[i]==client1Id) {
24                 //Петър Костов
25                 System.out.println(names[i]+" "+surnames[i]);
26             }
27         }
28
29         int client2Id = 10014; //В реална програма номерът
30 би
31                                     //идвал например от уеб
32 заявка
33         for (int i=0; i<clientNumbers.length; i++) {
34             if(clientNumbers[i]==client2Id) {
35                 //Ана Казакова
36                 System.out.println(names[i]+" "+surnames[i]);
37             }
38         }
39     }

```

Разпечатка 1. Програма, която не използва методи или механизми на обектно-ориентираното програмиране

На редове 6-13 са дефинирани масивите, които съдържат личните данни на клиентите. Читателите следва да забележат как са организирани тези масиви. Тъй като отделните информационни атрибути на клиентите са от различен тип, в програмата е създаден масив за всеки отделен атрибут. Данните на всеки клиент са разположени под един и същи индекс във всеки масив. Така например клиентският номер на Иван Янков е разположен на позиция 0 в масива **clientNumbers**. На позиция 0 във всички други масиви са разположени и името, презимето и фамилията на същия клиент.

За да намерим данните на даден клиент по клиентски номер, ние трябва да разберем просто на коя позиция в масива **clientNumbers** се намира този

клиентски номер. Това именно е предназначението на цикъла на редове 18-23, който търси позицията на клиентски номер **10303**, след което отпечата името и фамилията, разположени на същата позиция в масивите **names** и **surnames**.

Ако програмата свършваше тук, трудно можем да твърдим че нейният код има никакви сериозни дефекти. За добро или лошо обаче, реалните програми работят с много повече информационни атрибути. Кое е по-важно – те ги достъпват многократно от различни части на програмата по най-различни поводи. За да симулираме такава реална програма, на редове 25 до 32 ние правим още едно търсене, този път по различен клиентски номер.

С тази добавка не се изисква дори внимателно вглеждане, за да забележим, че имаме някакъв проблем. Цикълът на редове 27 до 32 повтаря до голяма степен кода на цикъла на редове 18-23. Единствената разлика е променливата, която носи информация за това какъв е клиентският номер на търсения клиент. Като вземем предвид колко често в реалните програми ние ще извършваме подобни търсения, става веднага ясно, че трябва да има начин повторението на редове 27-32 да бъде избегнато. Читателите едва ли ще бъдат изненадани когато заявим, че *методите* са създадени точно затова. Все пак ние използваме методи от самото начало на нашия курс. Така например всеки път, когато трябва да отпечатаме нещо на конзолата, ние използваме метод от стандартната библиотека на Java, който извикваме с името **System.out.println**. Този метод е написан веднъж, но ние модифицираме поведението му всеки път когато го извикваме като за целта му подаваме точната стойност, която трябва да бъде отпечатана. Методът **main** също е познат на читателите. Java го използва за входна точка на всички програми, поради което ние го декларирахме във всяка една програма досега. Време е да декларираме собствени методи и да се запознаем по-подробно с това как те се използват в Java.

Започваме с това, че за да се създаде нов метод, той трябва да бъде деклариран. Декларацията съдържа програмния код на метода, заедно с други задължителни и опционални елементи. Те са:

1. Ключовите думи **final**, **abstract** и **static**;
2. Ключовите думи **private**, **public** и **protected**, известни още като „модификатори за достъп“;
3. Списък с типови параметри, оградени от остри скоби;
4. Тип на данните, които се връщат от метода /задължителен елемент/;
5. Име на метода /задължителен елемент/;
6. Списък с параметри, оградени от кръгли скоби /задължителен елемент/;
7. Ключовата дума **throws**, следвана от списък с изключения, генерирани от метода
8. Блок от код, съдържащ декларациите, от които се състои метода;
9. Ключовите думи **final**, **abstract** и **static**, както и модификаторите за достъп са опционални. Техният смисъл ще бъде обяснен по-нататък.
Когато се използва ключовата дума **abstract**, методът няма блок от код.

Типовите параметри също са незадължителен елемент от дефиницията на класа и се използват за създаване на т.нар. „**джереник**“ (или още „**шаблонни**“) методи, които също ще разгледаме по-нататък.

Също опционален е и списъкът с изключения, които могат да бъдат генерирани от работата на метода. Работата с изключения също ще бъде разгледана по-нататък в изложението.

Долният пример съдържа декларации на методи, които имат само основните задължителни елементи и ключовата дума **static** (вж. разп. 2).

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     static double approximatePi() {
6         return 22.00/7.00;
7     }
8
9     static void printHeader() {
10        System.out.println("+-----+-----+-----+");
11        System.out.println("|Клиент № |Име      |Презиме  |Фамилия  |");
12        System.out.println("+-----+-----+-----+");
13    }
14
15    public static void main(String[] args) {
16        double halfPi = approximatePi()/2;
17        printHeader();
18    }
19 }

```

Разпечатка 2. Дефиниране на прости методи в Java

В разпечатка 2, освен методът **main**, програмистът дефинира и два допълнителни метода. На редове 5 и 9 потребителят дефинира съответно методи с имена **approximatePi** и **printHeader**¹⁴. И в двете декларации непосредствено преди имената на методите се изписва типът на връщаната от метода стойност. Връщаната стойност е задължителен елемент от всяка декларация на метод и не може да бъде изпускана.

Тук е моментът да споменем, че в програмирането различаваме два типа методи. Има методи, които връщат стойност, и методи, които не връщат стойност. Тази класификация е далеч от това да бъде „изкуствена“. Всъщност тя отразява дълбоките разлики в двата начина, по които ние използваме методите. От една страна ние използваме методи за извършването на изчисления. Такива методи приличат на математическите функции като синус и косинус, тъй като те връщат резултатна стойност. **approximatePi** в нашия пример е такъв метод. Връщаната

¹⁴ Всеки валиден идентификатор в Java може да бъде избран за име на метод, но по принцип добрият стил на програмиране предполага методите да имат имена, съставени от думи, като първата дума във всеки метод трябва да бъде глагол. Декларираните в примера методи не правят изключение, тъй като **“approximate”** и **“print”** са глаголи.

от него стойност е приближение на числото π . Тази стойност е от тип **double**, което е декларирано на ред 5.

Съществуват обаче и методи, чието единствено предназначение е да манипулират състоянието на определени обекти или пък състоянието на самия компютър. Такива методи обикновено не връщат стойност. В такъв случай при Java вместо тип на връщаната стойност се записва ключовата дума **void**. Методът **printHeader** е класически представител на тази категория методи. Освен извеждане на конзолата, методът не извършва никакви други действия и не връща резултат. Читателите вече са се сблъскали с подобни методи множество пъти. Методът **main** – входната точка на всяка програма – не връща стойност, поради което също се декларира с тип **void**. Със същия тип на връщаната стойност в стандартната библиотека е деклариран и **System.out.println**.

Методите се извикват чрез своето име, следвано от кръгли скоби, които съдържат евентуални параметри. Скобите са задължителни както при декларацията на метода, така и при извикването. Нашите два метода не вземат параметри, поради което и при декларациите, и при извикванията след имената на метода се изписват празни кръгли скоби.

Когато методът е деклариран с тип на връщаната стойност, той задължително трябва да върне такава. Това се извършва с ключовата дума **return**, следвана от самата стойност. **return** прекратява изпълнението на метода. Методите, които връщат **void**¹⁵, приключват при достигане на края на блока си от код. Опционално програмистът може да избере да извика **return**, което също прекратява метода.

Когато даден метод бива извикан, изпълнението на програмата преминава към неговия блок с код. Когато блокът с код приключи или бъде извикан **return**, изпълнението се връща към точката от програмата, от която е бил извикан метода.

¹⁵ Ние често ще казваме, че методът връща **void** за методите, които не връщат стойност, тъй като това опростява изложението.

Методите, които връщат стойност могат да участват в изрази. Така например на ред 16 **approximatePi** е част от израз за присвояване. Когато изпълнението стигне до извикване на метода, неговият код ще бъде изпълнен, той ще върне стойност и тази стойност ще участва в израза. Като краен резултат от изчислението на израза в променливата **halfPi** ще бъде поставено грубото приближение¹⁶ на числото $\pi/2$.

Методите, които не връщат стойност не могат да участват в изрази и се извикват като самостоятелна декларация, както е показано на ред 17.

За да приключим с разпечатка 2, ще направим едно последно наблюдение. Читателите трябва да забележат, че използването на методи (и особено на методи с добре избрани имена) прави програмния код по-четлив. Дори читатели, незапознати с методите за получаване на приближения на π , веднага биха разбрали какво прави методът **approximatePi** само по неговото име на ред 16.

Методите нямаше да са особено полезни, ако ние не можехме да модифицираме поведението им, подавайки им различни начални (или още „входни“) данни. За щастие методите позволяват подаването на списък с параметри, който изпълнява точно тази задача. В най-простата си форма, дефиницията на параметър се състои от тип и име на параметъра, което може да бъде всеки валиден идентификатор в Java. Когато методът има множество параметри, те се разделят със запетая. Долният пример демонстрира декларирането и извикването на метод, който има два параметъра:

¹⁶ Както и друг път сме отбелязвали, изчисляването на приближения на числото π е много удобно като малък учебен пример по програмиране. Начинът, по който е направено това в примера, е ужасен от гледна точка на прецизната математика. Ако извършваме сериозни изчисления, трябва да използваме далеч по-точното приближение, достъпно в Java чрез константата **Math.PI**.

```

1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     static double computeVolumeOfCylinder(double radius, double height) {
6         return Math.PI*radius*radius*height;
7     }
8
9     public static void main(String[] args) {
10        System.out.println("Обемът на цилиндър с радиус на основата "
11            +2+" и височина "+8+" е "
12            +computeVolumeOfCylinder(2,8));
13    }
14 }
15

```

Разпечатка 3. Извикване на метод с предаване на параметри в Java

Както е видно от разпечатка 3, декларацията на параметри на методите не е много по-различно от дефинирането на променливи. Параметрите на **computeVolumeOfCylinder** са два. **radius** е от тип **double** и **height** е от тип **double**.

В самия метод параметрите се използват по начин, който е напълно еквивалентен на начина по който се използват променливите. Така например в нашия пример двата параметъра се използват в израз на ред 6.

Ще използваме момента да проследим изпълнението на програма с методи още веднъж. Изпълнението на програмата от разпечатка 3 започва, разбира се, от метода **main**. Първата декларация от блока с код към този метод се намира на редове 10-12. Тя е извикване на метода **System.out.println**. Стойността за отпечатване е израз, който включва извикването на друг метод – дефинирания по-рано **computeVolumeOfCylinder**. Последният връща стойност, поради което използването му в израз е валидно.

На този етап изчислението на израза на редове 10-12 ще бъде временно спряно и изпълнението ще премине към **computeVolumeOfCylinder**. Методът **computeVolumeOfCylinder** получава два параметъра, зададени на ред 12. Първата подадена стойност е 2. Тя се поставя в параметъра **radius**, който е първият деклариран параметър в списъка с параметри. Втората подадена стойност е 8, което означава, че тя се поставя във втория параметър – **height**. Сред

като параметрите са получили своите стойности, изпълнението на метода **computeVolumeOfCylinder** може да започне. Блокът от код на метода се състои от един ред, който изчислява израз (ред 6). Този израз пресмята обема на цилиндър по стандартната формула и връща резултат чрез метода **return**. За конкретните параметри резултатът е числото 100.53096491487338.

Едва сред сега изразът на редове 10-12 може да бъде изчислен, като **computeVolumeOfCylinder(2,8)** ще бъде заменен с 100.53096491487338. След изчисляването на този израз, който всъщност слепва стойността с обяснителен текст, най-накрая ще бъде извикан **System.out.println**, който ще отпечата долупоказания резултат.

Обемът на цилиндър с радиус на основата 2 и височина 8 е 100.53096491487338

Методът **main** съдържа само тази декларация и след нейното изпълнение той приключва, с което приключва и програмата.

Читателят следва да забележи, че стойността, която връща методът **main** е **void**, поради което програмистът не е задължен да го завърши с **return**. От друга страна методът **computeVolumeOfCylinder(2,8)** връща **double** поради което задължително трябва да бъде приключен с **return**.

Извикването на методи има многобройни и сложни особености, които ще бъдат постепенно разяснени във връзка с използването на класове и обекти, но изложеният материал вече позволява да подобрим програмата от разпечатка 1.

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     static int findClient(int[] clientNumbers, int clientNumber) {
6         for (int i=0; i<clientNumbers.length; i++) {
7             if(clientNumbers[i]==clientNumber) {
8                 return i;
9             }
10        }
11        return -1;
12    }
13
14    public static void main(String[] args) {
15        String[] names = {"Иван", "Мария", "Петър", "Ана", "Марин",
16
```

```

17         "Катерина"};
18     String[] middleNames = {"Янков", "Петрова", "Александров",
19                             "Жекова", "Дончев", "Иванова"};
20     String[] surnames = {"Иванов", "Петрова", "Костов", "Казакова",
21                          "Петков", "Манева"};
22     int[] clientNumbers = {10013, 10201, 10303, 10014, 10005, 10006};
23
24     int client1Id = 10303; //В реална програма номерът би идвал
25                          //например от конзолата
26     int clientNumber = findClient(clientNumbers, client1Id);
27     //Петър Костов
28     System.out.println(names[clientNumber]+" "+surnames[clientNumber]);
29
30     int client2Id = 10014; //В реална програма номерът би идвал
31                          //например от уеб-формуляр
32     //Ана Жекова
33     clientNumber = findClient(clientNumbers, client2Id);
34     System.out.println(names[clientNumber]+" "+surnames[clientNumber]);
35 }

```

Разпечатка 4. Подобрения в програмата от разп. 1 с използване на методи

Програмата на разпечатка 4 подобрява програмата от разп. 1 с използване на методи. В подобрената програма функционалността по търсене на клиент по клиентски номер е изнесена в отделен метод. Този метод се нарича **findClient** и взема два аргумента – масив с клиентски номера и клиентски номер, чиято позиция в масива трябва да бъде намерена.

Първите и незабавни наблюдения върху нашия метод са свързани с това, че той като че ли не намаля кода на програмата. Това може да е вярно за нашата примерна програма, която замени два цикъла по 6 реда с един метод от 7 реда и две извиквания от по един ред, но няма да бъде вярно за една истинска програма, която може да извиква този метод значително повече пъти. Ние обаче далеч не сме приключили с оптимизациите. С постепенното усвояване на функциите на стандартната библиотека и обектно-ориентираното програмиране, ние ще сведем кода по същество до един – единствен ред.

Важно наблюдение също е това, че за разлика от предния пример ние подадохме променливи вместо литерали като параметри. Това е напълно допустимо. **За параметри на методите могат да бъдат подавани всякакви валидни изрази в Java.**

Това обаче води до по сериозен въпрос. Методът **findClient** се нуждае от масива **clientNumbers**, поради което ние го подаваме всеки път, когато извикаме метода. Няма ли начин **findClient** да използва променливата на масива директно? Краткият отговор на този въпрос е отрицателен. **Методите в Java не могат да достъпват променливи, дефинирани в други методи.** Но това далеч не обхваща всички особености на достъпа до променливите от страна на метода. Ние ще можем да ги разгледаме, едва когато покрием всички видове променливи в тази и следващите няколко глави.

Завършваме с някои наблюдения върху дефектите на новата програма. Въпреки че отделянето на повтарящия се код в метод подобри малко читаемостта на нашата програма, програмата все още е крехка и има два основни недостатъка. **На първо място** това, че всеки информационен атрибут трябва да се достъпва с индекс в съответния масив, изключва възможността да използваме **for-each** цикъл за търсене. Това пък на свой ред поражда реална опасност ние да объркаме индексите и прави нашия код податлив на трудни за откриване и отстраняване грешки по време на изпълнение на програмата. **Вторият проблем** се корени в това, че отделянето на функционалността за търсене в отделен метод поставя въпроса за това как методът следва да сигнализира за грешка при търсене, каквато например ще възникне, ако няма клиент с търсения клиентски номер. Избраното решение в случая е връщането на **-1**, което е невалидна позиция в масива. Смесването на валидни стойности с невалидни стойности, сигнализиращи грешка, е лоша практика, която в традиционните езици като C е просто неизбежна. Тъй като обхватът на смислените стойности, които се връщат от всеки метод, са специфични за метода, сигнализирането на грешка по този начин не може да бъде стандартизирано. Освен това е много лесно програмистът да изпусне да провери кода за грешка, точно както това е изпуснато в програмата от разпечатка 4. Ако се окаже, че търсеният клиентски номер липсва, това ще произведе грешка по време на изпълнение на програмата.

Както ще видим по-нататък, обектно-ориентираното програмиране представя ефективни решения и на двата проблема.

2. Използване на методи в Java

Както споменахме в предната точка, използването на методи има много тънкости и особености. Ние ще започнем да излагаме тези особености постепенно. Този процес ще започне в настоящата точка и ще приключи когато разгледаме обектите, класовете, интерфейсите и ламбда функциите в следващите глави. За улеснение на читателя ще опитаме да номерираме разгледаните в настоящата точка особености.

1. Всички параметри на методите в Java се подават по стойност

Завършихме предната точка с наблюдението че променливи от един метод не могат да достъпват директно променливи, дефинирани в друг метод. В същото време видяхме, че променливите от един метод *могат* да се подават на друг като параметри. Каква тогава е връзката между тези променливи и съответните параметри? Ако променлива е продадена като параметър и методът промени този параметър това значи ли, че и променливата ще се промени?

Различните програмни езици решават този проблем по различен начин. Подходът на Java обаче е напълно унифициран. **При предаване на една променлива като параметър, Java прави копие на стойността на променливата и го поставя в параметъра. Това означава, че ако методът промени параметъра, това не засяга стойността на оригиналната променлива.**

Трябва обаче да се помни, че в Java имаме променливи от елементарен и референтен тип. Предаването на променливи от референтен тип като параметри означава, че параметърът получава същата референция към оригиналната структура (обект или масив) в хийпа на програмата. **Ако методът, който бива извикан, използва препратката в референтния тип, за да промени обекта в**

хийпа на програмата, това ще засегне и оригиналния метод, тъй като и двата метода работят с една и съща препратка към една и съща структура.

Тъй като в Java много често се работи с референтни типове, горният факт е много важен и заслужава да се демонстрира с подробен пример:

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     static void modify(int e, int[] test) {
6         e=12;
7         test[0]=12;
8         test = new int[]{1,2,3};
9     }
10
11     public static void main(String[] args) {
12         int elementary = 0;
13         int[] arr = {0, 0, 0, 0, 0};
14
15         System.out.println(elementary); //0
16         System.out.println(arr[0]); //0
17
18         modify(elementary, arr);
19
20         System.out.println(elementary); //0
21         System.out.println(arr[0]); //12
22         //System.out.println(e); //Грешка по време на компилация
23     }
24 }
```

Разпечатка 5. Предаване на променливи от елементарни и референтни типове като параметри на методи

След стартиране на програмата, редове 12-13 инициализират две променливи. Първата от тях е от елементарен тип **int** и получава стойност **0**. Втората променлива е от референтен тип и сочи към масив от цели числа, инициализиран с нули. Редове 15 и 16 изрично отпечатват променливата **elementary** и първия елемент на масива, за да се потвърди тяхната нулева стойност.

Следва извикване на метода **modify** на ред 16. Тъй като Java предава всички параметри по стойност, стойностите на променливите **elementary** и **arr** се копират в параметрите **e** и **test**, след което изпълнението преминава към първия ред на блока от код на метода **modify**. Това е ред 6. Декларацията на този ред

променя стойността на параметъра **e**. В този момент **e** вече е 12, но тъй като в **e** има копие на стойността на **elementary**, променливата **elementary** не се променя.

Ред 7 е най-важната част от примера. Там променливата **test** се използва за да се достъпи масив от тип **int**. Тъй като масивите са референтни типове и тъй като в **test** има копие на референцията, която се намира в **arr**, и двете променливи сочат към един и същ масив в хийпа на програмата! Първият елемент на този масив получава стойност 12.

Декларацията на ред 8 се намира там за да демонстрира още веднъж разликата между самите референтни променливи и обектите, които са сочени от тях, както и да демонстрира, че на практика параметрите са променливи и се използват точно като такива. До ред 7 стойността на **test** бе равна на стойността на **arr**, тъй като **arr** бе подадена за стойност на параметъра **test** при извикването на метода. Нищо не пречи обаче методът да измени стойността на **test**, като присвои на променливата референция към нов масив. Точно това е направено на ред 8. Оттук нататък само **arr** сочи към оригиналния масив. Това още веднъж показва, че параметрите са просто малко по-особени променливи, които получават началната си стойност при извикване на метода.

С приключването на метода параметрите **test** и **e** се изчистват от стека на програмата и не са достъпни за извикващия метод. Опитът да се достъпят тези параметри извън метода **modify** води до грешка при компилация, каквато ще се получи ако премахнем коментарите от ред 22.

Редове 20 и 21 са поставени в примера, тъй като демонстрират описаните трансформации на стойностите на променливите. Тъй като модифицирахме оригиналния масив, неговият нулев елемент сега има стойност 12, докато стойността на променливата **elementary** остава непроменена.

Читателите без съмнение вече забелязват някои опасности при предаването на стойности като параметри. Метод, който променя състоянията на много от своите параметри, може да бъде доста опасен, тъй като програмистът може да забрави за част от неговите ефекти и да го извика неправилно. Ако читателите

изпаднат в подобна ситуация, би следвало да помислят за разделянето на метода на повече отделни методи.

Читателите вероятно също са забелязали че „животът“ и „видимостта“ на променливите в Java е ограничен и на този етап вероятно се питат какви са правилата, които ги определят. Ние ще разгледаме този въпрос скоро, когато покрием всички видове променливи.

2. Java не позволява задаването на стойности по подразбиране за параметрите

Както вече видяхме, Java позволява задаване на начални стойности на декларираните променливи. Задаването на стойности по подразбиране за параметрите обаче не е позволено. Долният хипотетичен код не е валидна Java програма:

```
1 //Хипотетична програма. Кодът не е валиден в Java!
2
3 package uk.co.lalev.javabook;
4
5 public class Main {
6
7     static void compute(int e, int p = 4) {    //!< Не е позволено в Java
8         return e*p;
9     }
10
11     public static void main(String[] args) {
12         compute(e); //!< Не е позволено в Java
13     }
14 }
```

Разпечатка 6. В Java не е позволено задаването на стойност по подразбиране за параметри

Двете фундаментални грешки в програмата на разп. 6 са обвързани. На ред 7 ние опитваме да зададем стойност за **p** по подразбиране. Езиците, които позволяват това, позволяват и изпускането на параметри, когато методът бива извикан.

В Java обаче, както вече споменахме, задаването на стойности по подразбиране не е позволено, поради което всички параметри задължително

трябва да бъдат подадени, когато методът бъде извикан. Това е и грешката на ред 12, където вторият параметър е изпуснат.

3. Методите в Java могат да бъдат дефинирани така, че те да приемат променлив брой параметри от един тип

Без да подозираме, ние вече използвахме утвърдения в Java начин за предаване на множество параметри от един тип към метод. Още в една от първите ни примерни програми, посветени на методите, ние предадохме променлива от тип масив на метод. Тъй като тази променливите от тип масив могат да сочат към масиви с произволен брой елементи, те са идеален механизъм за предаване на произволен брой параметри към метод.

Поставянето на параметри в масив, за да бъдат подадени на метод, обаче е неудобно, както това е илюстрирано в следващия програмен сегмент. Този сегмент съдържа метода `average`, който изчислява средноаритметична стойност на произволен брой числа:

```
6 static double average(double[] numbers) {
7     double sum=0;
8     for (int i=0; i<numbers.length; i++) sum+=numbers[i];
9     return sum/numbers.length;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(average(new double[] {0,1,2,3,4,5,6,7,8,9})); //4.5
14     System.out.println(average(new double[] {2,4,6})); //4.0
15 }
16
```

Разпечатка 7. Предаване на променлив брой параметри чрез използване на масив

Java предлага „синтактично улеснение“, което е илюстрирано на долната разпечатка, която изчислява средноаритметична стойност.

```
6 static double average(double... numbers) { //Променено
7     double sum=0;
8     for (int i=0; i<numbers.length; i++) sum+=numbers[i];
9     return sum/numbers.length;
10 }
11
12 public static void main(String... args) {
13     System.out.println(average(1,2,3,4,5,6,7,8,9,0)); //Променено
14     System.out.println(average(2,4,6)); //Променено
15 }
```

Разпечатка 8. Предаване на променлив брой параметри чрез използване на масив

При дефиниране на метод *последният* от параметрите може да бъде дефиниран както е показано на ред 6 на разп. 8. Както се вижда от ред 8, параметърът, деклариран по този начин, на практика е масив и се използва точно като масив (имащ поле **length** и индекси на елементите).

Истинското удобство обаче идва при извикването на метода. Също като при инициализаторите на масиви, където Java позволи заместването на конструкции като **new int[] {1,2,3}** просто с **{1,2,3}**, тук Java позволява изпускането на същия тип конструкции, базирани на ключовата дума **new**. Подаването на стойностите става директно, като те са разделени със запетаи (редове 13 и 14).

Читателят трябва да забележи също, че този път декларирахме метода **main** по същия начин с използването на многоточие. Това е напълно валидно, тъй като Java ще постави всички параметри, подадени от потребителя, в масива **args**, а използването на многоточие е просто алтернативен начин на задаване на масив, който може да бъде използван за последния от всички параметри.

Методите, декларирани в следващия програмен сегмент ще предизвикат грешки при компилиране на програмата, тъй като многоточието не е позволено при елементи, които не са последни в списъка с аргументи:

4	//Невалидни декларации на методи
5	static void computeVat(double ... itemPrices, double rate) {};
6	static double ... computeVat(double vatRate, double ... itemPrices) {};

Разпечатка 9. Предаване на променлив брой параметри чрез използване на масив

Декларацията на ред 5 е грешна, тъй като след декларирания с многоточие масив имаме нормален параметър. На ред 6 местата на нормалния параметър и масива с променливите параметри са коректни, но многоточие е използвано за типа на връщаната стойност от метода. Многоточието е позволено само за последния параметър от списъка с параметри. За да е коректен, методът на ред 6 трябва да замени дефиницията на връщаната стойност **double...** с **double[]**.

4. В Java могат да бъдат дефинирани методи, които имат еднакви имена, но различен списък с параметри

В терминологията на програмните езици това свойство се нарича „**овърлоудинг на методи**“¹⁷. Поради липса на по-добър превод на български, авторът често ще нарича резултатните методи „**вариантни**“ или „**алтернативни**“ методи вместо „**овърлоуднати методи**“, макар че не е убеден, че тази конвенция е стандартна. Долната разпечатка демонстрира как се дефинират и извикват вариантни методи в Java.

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4
5     static int getLeastSignificantDigit(String number) {
6         return number.charAt(number.length()-1)-'0';
7     }
8
9     static int getLeastSignificantDigit(int number) {
10        return number%10;
11    }
12
13    public static void main(String... args) {
14        System.out.println(getLeastSignificantDigit("395")); //5
15        System.out.println(getLeastSignificantDigit(106)); //6
16    }
17 }
```

Разпечатка 10. Овърлоудинг на методи

В разпечатка 10 има два метода с еднакво име и различен списък с параметри. Методът на ред 5 взема един параметър от тип **String**, докато методът от на ред 9 взема един параметър от тип **number**. Java преценява кой от двата метода да извика на база подадените параметри.

На ред 14 методът `main` извиква `getLeastSignificantDigit` с параметър от тип **String**. Този списък с параметри отговаря на първия от двата метода (редове 5-7), поради което именно той ще бъде изпълнен. На ред 15 методът `main` този път извиква `getLeastSignificantDigit` с параметър от тип **int**, което води до извикване на втория метод с това име (редове 9-11).

Използването на вариантни методи обикновено се прави от съображения за удобство, както е направено и в нашия пример. И двата метода намират първата

¹⁷ От англ. „**method overloading**“.

цифра на число. Първият метод работи директно върху числа, получени като символен низ (например от клавиатурата или от уеб заявка), докато вторият работи върху числа от тип **int**. Програмистът би могъл да използва и само един метод, но тогава преди извикването на метода трябваше да извърши конвертиране на единия тип в другия.

Използването на вариантни методи позволява и емулиране на параметри по подразбиране, каквито, както по-горе обяснихме, няма в Java. Долният пример демонстрира подобна употреба при функции, които изчисляват ДДС.

```
6      static double computeVAT(double vat, double[] prices)
7  {
8      double          sum          =          0;
9      for (double price : prices) sum+=price*vat;
10     return          sum;
11 };
12
13 static double computeVAT(double[] prices) {
14     return          computeVAT(0.2,          prices);
15 };
16
17 public static void main(String... args) {
18     double[] prices = {10.0, 20.0, 30.0, 40.0};
19     // ще извика computeVAT(double vat, double[] prices)
20     System.out.println(computeVAT(0.08, prices));
21
22     // ще извика computeVAT(double[] prices)
23     System.out.println(computeVAT(prices));
24 }
```

Разпечатка 11. Овърлоудинг на методи

Въпреки че горният програмен сегмент не е особено реалистичен, той добре демонстрира използването на вариантни методи за „емулирането“ на опционален параметър.

Извикването на **computeVAT(prices)** на ред 22 се извършва без посочването на процент на ДДС. Този метод обаче не прави нищо друго освен да извика на свой ред **computeVAT(0.2, prices)**, „попълвайки“ по този начин стойността за ДДС със „стойност по подразбиране“.

Читателите следва да запомнят този своеобразен „шаблон“, тъй като той се повтаря многократно. На практика навсякъде, където разработчиците на Java искат да позволят даден метод да се извиква със стойност по подразбиране, това се извършва по показания току-що начин чрез деклариране на два или повече вариантни метода.

5. При извикване на вариантни методи, Java решава кой метод ще бъде извикан на база „сигнатурата“ на всеки метод, която представлява комбинацията от името на метода и списъка с *типовете* на неговите параметри. Тази комбинация трябва да е уникална в рамките на дадения клас.

Долният пример демонстрира грешки, свързани с опитите да се създадат овърлоуднати методи с еднаква сигнатура:

```
1 public class Main {
2     public static int add(int a) {
3         return a;
4     }
5
6     public static int add(int a, int b) {
7         return a+b;
8     }
9
10    public static float add(float a, float b) {           //Грешка
11        return a+b;
12    }
13
14    public static float add(float c, float d) {           //Грешка
15        return c+d;
16    }
17
18    public static double add(float c, float d) {          //Грешка
19        return c+d;
20    }
```

Комбинацията от името на метода **add** с един параметър от тип **int** е уникална за класа **main**, така, че Java ще я позволи. Следващият метод е алтернативна версия, която има същото име и два параметъра от тип **int**. Тази сигнатура също е уникална, поради което методът се компилира без проблеми.

Третият, четвъртият и петият метод обаче имат една и съща сигнатура. Разликата между третия и четвъртия метод е единствено в имената на параметрите. **Имената на параметрите обаче са несъществени при формулиране на сигнатурата. Java взема предвид само типовете на параметрите.**

Метод 5 илюстрира същия факт. Методът има същите типове параметри като предните два, но различен тип на връщаната стойност. Тъй като **връщаната стойност не е част от сигнатурата**, тя е еднаква с тази на предните два метода и по тази причина генерира грешки при компилация.

Изискването за уникална сигнатура на метода произлиза от начина, по който в Java *извикваме* методите. Когато извикваме даден метод, ние предаваме само име и списък от стойности. На база комбинацията от името на метода и типовете на подаваните стойности (тоест сигнатурата на метода) Java трябва да реши кой конкретен метод трябва да бъде извикан. Това обяснява защо не можем да имаме методи с еднакви по типове параметри, които имат различни имена.

В някои случаи не е възможно да бъде определено кой метод трябва да бъде извикан на база *връщаната* стойност, поради което връщаната стойност също не е включена в сигнатурата на метода. Това е и причината, поради което ние не можем да имаме методи с еднакви имена и еднакви типове на параметрите, но различни типове на връщаната стойност в един клас.

6. Java не позволява в методите да има декларации, които не могат да бъдат достигнати в хода на изпълнението.

Както вече споменахме, програмистите могат да прекратят изпълнението на метода чрез извикване на **return**. Нещо повече, благодарение на механизми за условно изпълнение като **if** и **case**, те могат да поставят **return** на множество места в метода по такъв начин, че при настъпване на определени условия да се изпълнява един конкретен **return** оператор. Недопустимо е обаче **return** да се поставя така, че декларациите след него в блока от код да не могат да бъдат достигнати при никакви условия. Компиляторът на Java открива всички такива ситуации и генерира грешки при компилация.

```
7 public static void main(String[] args) {  
8     if (args.length != 3) {  
9         System.out.println("Неправилен брой аргументи");  
10        return;  
11        System.out.println("Довиждане!"); // Грешка  
12    }  
13  
14    return; // Позволено, но излишно  
15 }
```

Разпечатка 13. При поставянето на **return** трябва да се внимава за декларации, които не могат да бъдат достигнати по никой път на изпълнение

В примера от разпечатката, декларацията **return** на ред 10 оставя код, който не може да бъде достигнат, поради което това генерира грешка на ред 11. Читателите следва да забележат, че независимо от това, че методът **main** връща **void**, използването на **return** е напълно легитимно, както споменахме по-горе. Ред 14 демонстрира, че ако желае, разработчикът може дори да завърши блока от код на метода с **return**, въпреки че в случая това е напълно излишно и проява на лош стил.

7. При методите, които връщат стойност, всеки възможен път на изпълнение трябва да води до return декларация или до код, който генерира изключения.

По същия начин, както следи за това в методите да няма декларации, които не могат да бъдат достигнати вследствие на използването на **return**, при методите, които връщат стойност, компилаторът ще следи всеки път на изпълнение да води до **return** с нужния тип параметър или до генерирането на

изключение, което сигнализира грешка. Читателите лесно могат да си представят защо това е важно, представяйки си метод, който реализира математическата функция `sin`. Тъй като методът **sin** ще се използва в изрази, ситуации, при които не е ясно каква стойност трябва да бъде върната, трябва винаги да генерират грешка. В противен случай виртуалната машина на Java просто няма да знае какво да постави на мястото на функцията, когато тя се извика като част от израз с параметри, които водят до ситуация на „неопределеност“.

7	public static boolean isBigger(int a, int b) {
8	if (a>b) return true ;
9	} //Грешка при компилация - „Missing return statement”

Разпечатка 14. Не всички пътища за изпълнение водят до **return** декларация, а методът е деклариран като връщащ стойност

Показаният кратък пример на разпечатка 14 демонстрира функция, която сравнява две числа и връща **true** ако първото е по-голямо от второто. Функцията обаче е написана неправилно, тъй като не специфицира каква стойност трябва да бъде върната от метода, ако логическата проверка на ред 8 върне стойност **false**.

Глава 8. Въведение в обектно-ориентираното програмиране

Обектно ориентираното програмиране (ООП) е набор от техники и добри практики за тяхното прилагане, които водят до избягване повторното писане на подобни сегменти програмен код.

Това кратко определение, макар и точно, не може да предаде голямото значение, което ООП има за модерния софтуерен дизайн. Правилното прилагане на техниките за обектно-ориентирано програмиране позволи на разработчиците да създават модулен и лесен за поддържане програмен код. Подобренията, които ООП донесе в тази посока, бяха наистина значителни и на свой ред доведоха до появата на по-сложни, по-големи и многофункционални програми, които трудно можеха да бъдат създадени с помощта на по-старите методи. Ето защо днес обектно-ориентираното програмиране е стандартна парадигма, която се поддържа от повечето важни езици за програмиране. Java е един от най-силно обектно-ориентираните езици, които се използват в наши дни, поради което изучаването на ООП е дори по важно за начинаещите програмисти на Java, отколкото за техните колеги, които изучават други програмни езици.

Въведението в обектно-ориентираното програмиране типично се извършва чрез представянето на механизмите, които стоят в неговата основа. Това са **инкапсулацията, скриването, наследяването и полиморфизма**. Тези механизми могат да бъдат оприличени на „градивни блокове“, от които се изграждат по-сложните техники на ООП.

Отношението между тези градивни блокове и по-сложните техники за обектно ориентирано програмиране е много подобно на това, което съществува между основите на програмирането и теорията на алгоритмите. Читател, запознат с основите на програмирането, може да разработва собствени алгоритми за решаване на даден проблем и без да е запознат с теорията на алгоритмите. По същия начин читател, запознат с горните механизми, може да създава работещи ООП програми. Но така както теорията на алгоритмите разглежда нетривиални

решения на трудни и често повтарящи се алгоритмични проблеми, така и техниките за ООП предписват специфични комбинации от споменатите градивни блокове, които отразяват най-добрите практики, свързани с ефективната организация на програмния код в различни ситуации.

При представянето на основите на програмирането в предните глави ние запознахме читателите с програмните примитиви без да влизаме в дискусия на теорията на алгоритмите. Подобен подход ще използваме и при представянето на обектно-ориентираното програмиране. В нашето изложение ние няма да отидем отвъд основните механизми на ООП и тяхната реализация на Java. След усвояването на тези основи, читателите, които имат намерението да станат максимално ефективни програмисти, следва да се насочат към литература, която разглежда т.нар. „шаблони за дизайн“ („design patterns“). Така те ще се запознаят с естеството и дълбочината на проблемите, свързани с прилагането на ООП в някои сложни реални ситуации.

1. Инкапсулация

Инкапсулацията при обектно-ориентираното програмиране касае синтеза на две много интуитивни идеи. Първата от тях касае групирането на няколко променливи, които носят информация за даден обект в реалния свят, в една съставна синтактична структура на езика Java. Групирането е много полезно, тъй като независимо дали обработваме данните с компютър или не, събирането на всички данни за даден обект от реалния свят на едно място води до по-голямо удобство и по-лесна обработка. В Java ние наричаме тази структура „клас“. Променливите, които са включени в един клас наричаме „полета“ на класа.

Втората идея е „окомплектоването“ на класовете с методи, които извършват действия върху данните, съдържащи се в полетата класа. Добавянето на методи към данните отива далеч отвъд простото групиране и позволява на програмистите да мислят по различен начин за програмата. Програмистите

мислят за класовете като един вид автомати, които могат сами да извършват определени действия върху данните, инкапсулирани в тях.

Така например клас в истинска програма, който представлява триъгълник, най-вероятно може и знае как да изчислява сам лицето си, тъй като програмистът, създаващ класа, е добавил метод за това. По същия начин клас, който представлява личните данни на български гражданин, вероятно ще може сам да определи дали стойността на подаденото за гражданина ЕГН е коректна.

Инкапсулацията на методи наистина блести, когато даден клас е написан от един програмист, а се използва от друг. Благодарение на включването на методи в класа, програмистът-ползвател не трябва да учи и да познава вътрешната структура на данните и променливите на класа, а е достатъчно да научи важните действия, които могат да се извършват с този клас. За класовете в реалните програми това е голяма стъпка напред, тъй като тяхната структура много пъти е доста сложна и се променя често поради изменения на изискванията към функционалността на програмата.

За да дадем пример за вътрешна структура на клас, можем да използваме триъгълника от предния пример. Той може да се реализира по няколко начина. В зависимост от нуждите и виждането на програмистите, триъгълникът може да бъде представен като координатите на три точки върху координатна система, или пък просто като дължините на трите страни на триъгълника, или дори само като дължината на една страна и два ъгъла. Програмистът-ползвател, който иска да получи лицето на триъгълника, не трябва да знае в детайли как вътрешно се съхраняват данните за триъгълника и как от тези данни се получава търсения резултат. Той просто трябва да знае кой метод на класа – триъгълник връща неговото лице. Именно това е същността на инкапсулацията.

При по-старите техники за програмиране, постигането на инкапсулация на такова ниво беше невъзможно, тъй като всеки метод работеше и съществуваше сам за себе си. При извикването на такъв „самостоятелен“ метод програмистът трябваше ръчно да подаде като аргумент всяка една променлива, която

представлява вътрешното състояние на обработвания обект. Поради това програмистът трябваше да знае в детайли как вътрешно се представят данните за всеки един обект, обработван от програмата.

Продължавайки в контекста на предния пример, метод в една класическа не-ООП програма, който пресмята лице на триъгълник, трябва да получи като аргументи данните за точките, трите страни или страната и ъглите, с които вътрешно се представя този триъгълник според виждането на програмиста – автор на метода. Програмистът-ползвател тогава трябва да знае точно предназначението на всяка променлива и ролята и във вътрешното представяне на триъгълника, за да може да подаде правилно входните данни.

1.1. Инкапсулация на полета

В минималната си форма, декларацията за създаване на нов клас в Java се състои от ключовата дума **class**, следвана от името на класа и блок от код, който съдържа декларации на променливи. Вече споменахме, че ние наричаме тези променливи „*полета*“.

Долната разпечатка демонстрира конкретно как се декларира клас:

```
1 package uk.co.lalev.javabook;  
2  
3 class Person {  
4     String name;  
5     String middleName;  
6     String surname;  
7     int age;  
8 }
```

Разпечатка 1 Деклариране на клас

Веднъж дефиниран, един клас може да се използва като тип, подобен на **int**, **String** или **double**. С това е свързано и първото важно разграничение, което начинаещите в ООП програмисти трябва да се научат да правят. Класовете типично не съдържат реални данни, а само дефинират структурата, в която реалните данни ще бъдат съхранявани. С други думи те дефинират нов

„тип“ данни. За да съхраним реалните данни, ние се нуждаем от променливи, които са от този тип. Ние наричаме тези променливи „**обекти**“ или понякога „**инстанции**“ на класа.

Читателите могат да мислят за класовете като за чертеж на сграда, а за обектите като сграда, построена по този чертеж. Чертежът казва колко етажа ще има сградата, колко обособени апартамента, търговски площи и пр. ще има на всеки етаж. По дадения чертеж обаче могат да бъдат направени множество реални сгради. Във всяка от тях на съответния етаж ще живеят различни хора, във всяка обособена търговска площ ще има различни магазини. По същия начин *класът* казва какви полета ще бъдат групирани заедно, а *обектът* от тип дадения клас съдържа реалните данни, които се съхраняват в паметта на компютъра.

Долната разпечатка разширява разпечатка 1 като към дефиницията на клас добавя и демонстрация как в Java се създават променливи от типа на класа:

```
1 package uk.co.lalev.javabook;
2
3 class Person {
4     String name;
5     String middleName;
6     String surname;
7     int age;
8 }
9
10 public class Main {
11     public static void main(String[] args) {
12         Person firstPerson;
13         firstPerson = new Person();
14         firstPerson.name="Петър";
15         firstPerson.middleName="Иванов";
16         firstPerson.surname="Георгиев";
17         firstPerson.age=43;
18
19         Person secondPerson = new Person();
20         secondPerson.name="Ана";
21         secondPerson.middleName="Ангелова";
22         secondPerson.surname="Ангелова";
23         secondPerson.age=29;
24     }
25 }
```

Разпечатка 2. Използване на класове

Редове 3-8 дефинират клас **Person** с няколко полета – **name**, **middleName**, **surname** и **age**. На ред 12 програмистът създава първата си променлива от тип

Person, но особено важната част на примера идва на следващия ред. Той използва ключовата дума **new**, за да създаде нов „екземпляр“ на класа **Person**, който да бъде присвоен на променливата **firstPerson**. Точното име за този екземпляр, разбира се е „*обект*“, а процеса по създаването му наричаме „*инстанциране*“.

Читателите, които са се запознали с главата за масивите, ще забележат сходство в синтаксиса. Това не е случайност, тъй като **в Java всички променливи, които са от тип някакъв клас, са референтни**. С други думи променливите, които имат за тип някакъв клас, са препратки към реалните обекти, които се намират в хийпа на програмата.

Точно като при масивите, създаването на променлива от тип даден клас не води автоматично до създаването на обект (т.е. инстанция на този клас). Ако с една декларация ние създадем само променлива от типа на дадения клас, както сме направили на ред 12 в примера, тази променлива няма да сочи реален обект, а ще съдържа специалната стойност **null**, която указва, че нашата препратка (т.е. референция) не сочи никъде. Едва щом присвоим на променливата реална референция (тоест направим така, че тя да сочи към реално създаден обект в хийпа на програмата) ние можем да я използваме за да достъпим полетата на обекта, както например е направено на редове 14-17 на примера.

За да „заредим“ променливата с реална референция ние можем да направим само две неща. Ние можем да създадем нов обект с ключовата дума **new** и да присвоим референция към него на нашата променлива, както е показано на ред 13 в примера. Другият вариант е да присвоим на нашата променлива стойността на друга променлива от същия тип, която вече съдържа препратка към реален обект.

Създаването на обект и присвояването на референция към него на новодекларирана променлива може да се извърши и на една стъпка, както е показано на ред 19 с променливата **secondPerson**.

Разликата между променливата и соченият от нея обект е именно и второто голямо разграничение, което начинаещите програмисти на Java трябва да се научат да правят.

Това, че променливите от тип клас са референтни, означава, че също както при масивите можем да имаме две или повече различни променливи, сочещи към един и същ обект. В този случай промяна, направена върху обекта чрез едната променлива, е мигновено видна чрез другата. Също така, при предаване на обект като параметър на метод, този обект не се копира, както се копират стойностите на елементарните променливи. Вместо това се предава референция към обекта и като резултат от това методът работи върху оригиналния обект.

Веднъж създадена, променливата от тип **Person** в нашия пример има всички полета, дефинирани по-горе в класа. Тези полета се достъпват като се изпише името на променливата, следвано от точка и името на полето. Така например на ред 16 програмистът присвоява стойност 43 на полето **age** на променливата **firstPerson**.

Читателите, съпоставящи казаното по-горе за инкапсулацията с разпечатка 2, ще отбележат, че примерът не демонстрира как програмистът-ползвател няма да се нуждае от познания за вътрешната структура на класа. Ние всъщност инициализирахме всяко едно поле с неговата стойност и по този начин демонстрирахме компетентност относно вътрешната структура на класа **Person**. Това е така, тъй като до момента ние сме представили твърде малко възможности на класовете. С представянето на нови и нови такива, в следващите примери ние ще подобрим значително инкапсулацията и ще демонстрираме нейната полезност.

1.2. Инкапсулация на методи

Читателите вероятно вече са убедени, че инкапсулирането на полетата в клас може да бъде полезно. Но инкапсулацията е повече от простото събиране на

информационните атрибути на един обект от реалния свят в една синтактична конструкция, каквато са класовете. Както уточнихме вече, класовете всъщност могат да съдържат и методи. Тук е моментът да споменем (вероятно за пореден път), че **в Java всъщност не можем да дефинираме методи, които се намират извън класове!**

Заължителното изискване всеки метод да се числи към даден клас, наложено от езика Java, бе и причината да създаваме класове във всяка една примерна програма досега, дори и в тези, които имаха само един елементарен метод. Поради липса на по-описателно име и поради това, че този клас съдържаше входната точка на програмата ни (методът **main**), често като единствен метод, повечето пъти ние наричахме този клас **Main**.

Читателите могат да разгледат следващата разпечатка и да се убедят във това, че макар методите да са повече от един, при това разпределени между двата класа в програмата, в техните дефиниции няма нищо, което да не е срещано в предходни примери.

```
1  class Triangle {
2      double sideALength;
3      double sideBLength;
4      double sideCLength;
5
6      /**
7       * Изчислява лице на триъгълник по формула, включваща трите страни на
8       * триъгълника.
9       */
10     double computeArea() {
11         double semiPerimeter = (sideALength+sideBLength+sideCLength)/2;
12         return Math.sqrt(semiPerimeter *
13             (semiPerimeter-sideALength) *
14             (semiPerimeter-sideBLength) *
15             (semiPerimeter-sideCLength));
16     }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Triangle t = new Triangle();
22         t.sideALength = 3.0;
23         t.sideBLength = 4.0;
24         t.sideCLength = 5.0;
25         System.out.println("Лицето на триъгълник със страни 3.0, 4.0 и 5.0 е
26 "+t.computeArea());
27     }
28 }
```

Програмата от разпечатка 8.3 съдържа два класа, всеки от които с по един метод. Имаме класа **Triangle**, който реализира триъгълник. Класът има един метод – **computeArea**. Също така имаме и класа **Main**, който съдържа входната точка в програмата – методът **main**.

Методът **computeArea** е написан максимално реалистично, включвайки дори специалните Javadoc коментари (редове 6-9). Тези коментари могат да се разпознаят по двете звезди, следващи отварящата наклонена черта. С помощта на едноименния инструмент те могат да бъдат превърнати директно в HTML референтна документация. Всеки професионално написан клас има подобна документация, която е насочена особено към методите, които ще бъдат извиквани от програмистите-ползватели на класа.

Разпечатка 3 има един много важен нов елемент и той не е свързан с това как декларираме методи, а касае начина, по който смесваме локални променливи, методи и полета в класа.

В многото примери досега ние използвахме само променливи, дефинирани вътре в даден метод. Сега тепърва започваме да работим с полета, които са променливи, числящи се към класа, а не към конкретен метод. За да можем да съпоставим гладко свойствата на едните и на другите променливи, ние се нуждаем от по-прецизни имена. Още от самото им въвеждане, ние възприехме името „поле“ за променливите на класа. Оттук нататък ние ще използваме понятието „*локална променлива*“ за променливите, дефинирани вътре в даден метод. На практика всички променливи, които използвахме в примерите от предните глави са локални.

Долната таблица обобщава разликите между двата вида променливи:

Локални променливи	Полета
Съществуват и могат да бъдат достъпени само вътре в блока от код, в който са дефинирани. Най-често този блок от код е	Съществуват докато съответният обект съществува и не е освободен от виртуалната машина. Ако няма зададени други ограничения, достъпът до тях може да

тялото на метод, но може да бъде и тялото на цикъл или условна конструкция.	се осъществява както от вътрешни за класа на обекта методи, така и от външни за класа на обекта методи.
Съхраняват временна информация/междинни резултати, които са нужни по време на изпълнението на метода.	Съхраняват информация, която е нужна по време на целия живот на обекта. Понякога на високо ниво ние говорим за тази информация като „ състояние на обекта “.
Началните стойности на променливите трябва да бъдат зададени задължително от програмиста.	Ако програмистът не зададе начални стойности на полетата, те се попълват със стандартни стойности по подразбиране (например 0 за полета от тип int , 0.0 за полета от тип Double и false за полета от тип boolean , както и null за всички променливи от референтни типове, включително String)
Опит за използването на неинициализирана променлива води до грешка още при компилация на програмата.	Полетата не може да не са инициализирани, но в много случаи използването на поле без то да бъде инициализирано предварително от програмиста (тоест поле, което е попълнено със стойност по подразбиране) ще доведе до трудна за откриване логическа грешка в програмата.

Табл. 1. Разлики между полетата и локалните променливи

Може би най-объркващо за начинаещите програмисти при първото съприкосновение с полетата е това как трябва ги указваме, когато осъществяваме достъп до тях. Така например на ред 24 на разпечатка 3 ние използвахме **t.sideCLength** за да осъществим достъп до полето **sideCLength** на обекта, сочен от променливата **t**. Тук едва ли има някаква неяснота или дилема, тъй като ние се намираме в метода **main**, който се числи към друг клас. И тъй като обектите съдържат реалните данни, е ясно че ние се нуждаем от променлива, сочеща към реален обект от тип **Triange**, за да можем въобще да мислим за достъп до поле **sideCLength**.

Но точно от кой обект идва полето **sideCLength**, зададено на ред 15? То е посочено като локална променлива, без префикс, който да указва от кой обект трябва да дойде то.

Ред 15 се намира в метода **computeArea**, който се числи към същия клас, в който е дефинирано полето **sideCLength**. Използването на **sideCLength** без

променлива-префикс означава, че полето трябва да се вземе от текущия обект. Но кой обект е текущ? Отговорът е, че обикновените методи, които дискутираме до момента, винаги трябва да се извикват чрез конкретна инстанция на класа, в която е дефиниран метода. Именно този обект играе роля на текущ за метода.

Кратка справка с изходния код на примера показва, че методът **computeArea** е извикан само веднъж на ред 26 върху променливата **t**. Следователно изпълнението на **computeArea** ще бъде върху обекта **t**, който ще играе ролята на текущ за това извикване.

Разбира се, ако в пример 8.3 имаше друга променлива от тип **Triangle**, извикването на **computeArea** върху тази променлива щеше да доведе до това, че полетата, посочени в **computeArea**, щяха да бъдат взети от тази променлива.

1.3. Роля на конструкторите за постигане на инкапсулация

В предходната точка на няколко пъти споменахме, че инкапсулацията помага на програмистите да мислят за класовете в програмата като автоматизирани същности, които знаят как да изпълняват определени дейности върху себе си. За да могат да мислят на такова високо ниво, програмистите, които ползват даден клас, не трябва да бъдат задължавани да знаят детайли за вътрешната реализация на този клас, каквато например са полетата, съхраняващи данни. В същото време ние не успяхме да демонстрираме ефективна инкапсулация с разгледаните до момента инструменти.

Сега адресираме този проблем, представяйки следващия важен инструмент за постигане на инкапсулация. Това са специалните методи, наречени „**конструктори**“. Тези методи отговарят за правилното инициализиране на полетата на новосъздадените обекти.

Долната разпечатка разширява примера от разпечатка 3, като добавя конструктор към класа **Triangle**.

```

1  class Triangle {
2      double sideALength;
3      double sideBLength;
4      double sideCLength;
5
6      double perimeter;
7
8      double computePerimeter() {
9          return perimeter;
10     }
11
12     double computeArea() {
13         return Math.sqrt((perimeter/2) *
14             (perimeter/2 - sideALength) *
15             (perimeter/2 - sideBLength) *
16             (perimeter/2 - sideCLength));
17     }
18
19     Triangle(double newSideALength, double newSideBLength, double
newSideCLength) {
20         if ((newSideALength<0) || (newSideBLength<0) || (newSideCLength<0)) {
21             throw new RuntimeException();
22         }
23
24         sideALength = newSideALength;
25         sideBLength = newSideBLength;
26         sideCLength = newSideCLength;
27         perimeter = sideALength + sideBLength + sideCLength;
28     }
29 }
30
31 public class Main {
32     public static void main(String... args) {
33         Triangle t = new Triangle(3.0,4.0, 5.0);
34         System.out.println(t.computePerimeter()); // 12.0
35         System.out.println(t.computeArea());      // 6.0
36     }
37 }

```

Разпечатка 4. Дефиниране и използване на конструктор на клас

Програмата на разпечатка 4 отново надгражда предходната, като добавя и изменя елементи в класа **Triangle**. Модифицираният клас вече има два метода, както и метод-конструктор.

Методът **computePerimeter** връща обиколката на триъгълника, докато методът **computeArea** отново изчислява и връща лицето на триъгълника. Конструкторът се намира на редове 19 до 28 на разпечатката и може да бъде разпознат по това, че неговата дефиниция не съдържа ключова дума, която

показва връщания тип (дори нито **void**), както и по това, че неговото име съвпада с името на класа.

Конструкторите в Java не връщат стойности и винаги се казват по същия начин като класа, към който се числят. Освен тези особености, те са до голяма степен обикновени методи, които могат да вземат аргументи и дори могат да бъдат овърлоудвани (т.е. могат да се напишат различни варианти на метода с едно име, но различен набор от аргументи).

Както е видно от ред 19 на примера, нашият конструктор взема 3 аргумента, които представляват дължините на страните на триъгълника. Тези аргументи са съответно **newSideALength**, **newSideBLength** и **newSideCLength**.

Главната задача, която конструкторът ще изпълни, е присвояването на тези стойности на вътрешните полета на класа с имена **sideALength**, **sideBLength** и **sideCLength**. Преди това обаче, конструкторът ще извърши още една много важна задача, а именно валидацията на данните.

Валидацията на данните е процес при който програмата проверява дали нововъвежданите данни са логически валидни. Тъй като конструкторът е първият метод, който се извиква при създаване на обект от даден клас, това типично е мястото, в което се правят проверки за валидност на данните.

Нашият конструктор от разпечатка 8.4 ще провери дали случайно погрешка програмистът-ползвател не подава отрицателни стойности за дължини на страните (редове 20-22). Тъй като отрицателните стойности нямат смисъл като дължина на страни на триъгълник, ако конструкторът установи такава, той ще сигнализира за грешка чрез извикване на **throw**¹.

¹ Точният механизъм за сигнализиране на грешки в програмата е разгледан в една от следващите глави. Тук е достатъчно да споменем, че сигналът за грешка, генериран от **throw**, се нарича „изключение“ и по подразбиране ще спре изпълнението на програмата с предупредително съобщение, предотвратявайки по този начин по-сложни и непредвидими логически грешки, породени от извършване на последователност от операции върху грешни начални данни.

Едва след извършване на валидацията, конструкторът задава **newSideALength**, **newSideBLength** и **newSideCLength** за стойности съответно на полетата **sideALength**, **sideBLength** и **sideCLength** (на редове 24-26).

Освен това с последната си декларация на ред 47, конструкторът инициализира ново (в сравнение с предния пример) поле, което съдържа обиколката на триъгълника. В предния пример обиколката (по-точно половината от нея) се изчисляваше в локалната променлива **semiPerimeter**, разположена в метода **computeArea**. Модифицираният **Triangle** клас от текущия пример има методи, които връщат не само лицето, но и обиколката на триъгълника, поради което програмистът е преценил, че се нуждае от допълнително поле, което да съхранява тази обиколка, вместо да я изчислява два пъти в отделните методи.

Спирайки за момент дискусията на примера, най-накрая можем да сравним класовете с конструктор спрямо тези, които нямат такъв. Би трябвало да е видно, че класовете с конструктор имат сериозни предимства.

В оригиналния пример нищо не пречеше на програмиста да въведе негативни стойности за дължините на страните. Подобни стойности обаче нямат смисъл в геометрията и ще „счупят“ формулата за лицето на триъгълника. Благодарение на конструктора обаче, ние вмъкнахме валидация на входните данни. Така ако програмистът-ползвател допусне грешка при задаването на начални стойности, ние имаме допълнително ниво на защита, което ще предотврати по-сложни и трудни за откриване грешки. От гледна точка на програмиста-ползвател ще изглежда, че нашият обект автоматично валидира подадените входни данни.

Освен това, ние вече можем да видим ситуация при която от три входни стойности, ние инициализирахме 4 полета. Така програмистите ползватели на нашия клас **Triangle** няма нужда да знаят никога, че използваме поле, което вътрешно съхранява периметъра на триъгълника. Ако тези програмисти се въздържат да използват полетата на нашия клас директно, ние спокойно можем да реструктурираме класа **Triangle**, избирайки друга реализация и други полета.

Докато запазваме същия конструктор и същите методи, новият ни клас ще може да се използва по абсолютно непроменен начин. Това намаля нуждата от преписване на големи части от програмата всеки път когато променяме класа **Triangle**.

Конструкторът е важен и за виртуалната машина. Всъщност той е толкова важен, че **ако програмистът не дефинира конструктор за даден клас, Java автоматично ще създаде празен конструктор без аргументи**. Това означава, че трябва да внимаваме, когато използваме понятието „клас без конструктор“, както направихме по-горе, тъй като в действителност няма класове без конструктор. Стандартният конструктор, генериран по подразбиране, обаче не върши нищо конкретно от гледна точка логиката на програмата. Полезно е обаче програмистите да са наясно с неговото съществуване, тъй като това ще им помогне да изградят „мисловния модел“, който отговаря на механиката на инициализиране на сложните последователности от класове. Разбира се, ние ще разгледаме тази механика в следващите глави, когато сме усвоили основите на ООП.

След като се запознахме с това какво извършва конструктора на **Triangle**, остава да обясним как точно се извиква този конструктор.

В Java конструкторите се извикват при създаване на обект от даден клас чрез ключовата дума **new**. Нещо повече, техните аргументи се предават след името на създавания обект.

В пример 4 на ред 33 ние декларираме променлива от тип **Triangle** и веднага след това ѝ присвояваме новосъздаден **Triangle** обект. Читателите следва да забележат, че след **new** идва името на новосъздадения обект (което задължително съвпада с имената на всички дефинирани конструктори за този обект). Точно както при методите след това име се предават аргументите за конструктора на обекта, заградени в скоби. **Ако съществуват множество овърлоуднати версии на конструктора, Java избира кой конструктор да**

извика на база типовете на подадените в скобите стойности, точно както при обикновените овърлоуднати методи.

По-конкретно, извикването на **new** извършва две действия. На първа фаза виртуалната машина заделя място в хийпа на програмата за новия обект. След като тази фаза приключи, Java извика конструктора, който отговаря на подадените на **new** параметри. Тази втора стъпка ще доведе, разбира се, до попълване на полетата и правилната инициализация на новия обект.

В нашия пример трите стойности, които са подадени на ред 33 са 3.0, 4.0 и 5.0. Тези стойности ще станат съответно аргументите **newSideALength**, **newSideBLength** и **newSideCLength** на конструктора, откъдето, съгласно инструкциите на програмиста ще бъдат попълнени полетата на класа **Triangle**.

Връщайки се към пример 2 сега можем да обясним защо ключовата дума **new**, която в този пример създаваше инстанции на класа **Person**, имаше празни кръгли скоби след името на класа **Person**. Точно като при методите, които не вземат аргументи, празните скоби обозначават липса на аргументи, които да бъдат подадени на конструктора.

1.4. Пакети

Класовете в Java винаги се дефинират в **пакети**, което също е форма на инкапсулация. Пакетите съдържат набор от класове, които са създадени за извършване на някаква конкретна задача и са по някакъв начин свързани.

Така например класът, който представлява един студент, има множество информационни атрибути, като ЕГН, факултетен номер, специалност и т.н. Някои от тези атрибути обаче също са „съставни“ и трябва да се дефинират като отделни класове. Такъв атрибут например е оценката, която „носи в себе си“ числова оценка и оценка с думи, както и предмет, по който е написана. Самият предмет също може да е обект, тъй като той също има множество информационни атрибути, като например наименование, брой часове в учебната програма и т.н.

Класове, които силно зависят един от друг, се поставят в един пакет. Малките програми обикновено се състоят от един пакет, докато големите програми могат да са съставени от повече пакети.

Ние определяме кой клас се числи към даден пакет, чрез самите файлове, в които дефинираме класовете. В началото на всеки *файл* се поставя ключовата дума **package**, която е следвана от уникално име на пакета, избрано от програмиста. Всички класове, които са дефинирани в този файл, се считат за числящи се към този пакет.

Когато клас в един пакет трябва да използва клас от друг пакет, първият клас „**импортира**“ втория чрез декларацията **import**, която демонстрирахме множество пъти по-рано.

Долният пример е много кратка илюстрация на току-що споменатите механизми.

Student.java	
1	package com.someorg.Student;
2	
3	import java.util.Date;
4	
5	public class Student {
6	String ime;
7	String prezime;
8	String familia;
9	String egn;
10	Mark[] marks;
11	Date rojdennaData;
12	}
Mark.java	
1	package com.someorg.Student;
2	
3	public class Mark {
4	String predmet;
5	String tekst;
6	int cifra;
7	}

Разпечатка 5. Класове и пакети

В примера на разпечатка 5 класът **Student** представлява хипотетични студенти, а класът **Mark** – техните оценки. Класът **Student** съдържа различни

информационни атрибути, както и масив от оценки. Ето защо той се нуждае от класа **Mark**, който описва как трябва да „изглежда“ една оценка.

Двата класа могат да бъдат групирани в един пакет съвсем естествено, тъй като връзката между тях е доста тясна. Програмистът на даденото приложение едва ли може да използва класа **Mark** за нещо друго, освен като пояснение към класа **Student**. Ние можем да видим, че програмистът е отчел това и двата класа наистина са в един пакет. Това е видимо от първите редове на двата файла, които имат **package** директива с име на пакета, което е еднакво за двата файла.

Класът **Student** използва и друг клас. Това е класът **Date** от пакета **java.util** на стандартната библиотека на Java. **Date** представлява дати и е нужен, за да представим датата на раждане на студента. За разлика от **Mark**, **Date** е далеч по-универсален клас, който очевидно може да е полезен в много сценарии, поради което връзката му към **Student** не е толкова „тясна“.

Date е деклариран в пакета **java.util**, наред с подобни и много полезни „обслужващи“ класове. Ние можем да видим, че **Student.java** импортира класа **Date** на ред 3. Без декларацията на ред 3, компилаторът ще спре с грешка на ред 11, която ще обясни, че компилаторът не знае как да намери клас **Date**. Читателите трябва да забележат, че класът **Mark** не е импортиран по такъв начин. Това е така, тъй като **Mark** е в същия пакет като **Student**.

Пакетите, също като конструкторите, са задължителни. Ако програмистът пропусне да вмъкне **package** декларация на първия ред на файла, в който е дефиниран даден клас, този клас ще бъде поставен в пакета „по подразбиране“, който няма име. Такъв клас много трудно може да бъде споделен с други програмисти, ако това се наложи.

В нашите учебни примери ние често изпускаме **package** декларацията и поставяме класовете в пакета по подразбиране. Нашето извинение е, разбира се, че класовете на учебните ни примери едва ли ще бъдат част от някаква по-сложна програма, споделяна с други програмисти.

В следващите глави и точки ние ще изложим правилата за формиране на уникалните имена на пакетите, както и отношението им към другите механизми на ООП.

2. Наследяване

Наследяването е най-важният механизъм на обектно-ориентираното програмиране, тъй като позволява на един клас да разшири кода на друг клас, без да се налага да повтаря методи и дефиниции на полета, които са по същество еднакви за двата класа. Класът, който бива разширяван, често се нарича „*клас-родител*“, докато класът, който бива наследяван, съответно бива наричан „*клас-наследник*“. Ние често ще наричаме класа-родител – „*суперклас*“, а класът наследник – „*субклас*“. В понятията „суперклас“ и „субклас“ обаче ние често ще влагаме допълнителен смисъл. Тъй като ние можем да имаме „дълбока“ йерархия на наследяването, при която например клас А е родител на клас В, клас В е родител на клас С и клас С е родител на клас D, под „суперклас“ ние ще имаме предвид клас-предшественик, който се намира нагоре в йерархията на наследяването, без задължително този клас да бъде „пряк“ родител на отправния ни клас. В контекста на нашия пример клас D има само един клас-родител и това е класът С. Но D има много суперкласове, а именно А, В и С. Всички тези класове се намират „над“ D в йерархията на наследяването.

По същия начин под „субклас“ на даден клас ние имаме предвид клас, който е пряк или непряк наследник на дадения и се намира „под“ него в йерархията на наследяването.

Наследяването в Java се задава, като след името на класа-наследник се постави ключовата дума **extends** и се добави името на класа-родител.

Долният пример демонстрира как на практика се извършва това:

1	class Account {
2	double balance;
3	

```

4  void credit(double amount) {
5      balance = balance+amount;
6  }
7
8  boolean debit(double amount) {
9      if (amount<=balance) {
10         balance = balance-amount;
11         return true;
12     } else {
13         return false;
14     }
15 }
16
17 boolean withdraw(double amount) {
18     boolean success = debit(amount);
19     if (success) {
20         System.out.println("Successful withdrawal of "+amount);
21     } else {
22         System.out.println("Unsuccessful withdrawal of "+amount);
23     }
24     return success;
25 }
26
27 void deposit(double amount) {
28     System.out.println("Deposit of "+amount);
29     credit(amount);
30 };
31
32 Account() {
33     balance=0;
34 }
35 }
36
37 class NumberedAccount extends Account {
38     int number;
39
40     NumberedAccount(int newNumber) {
41         balance = 0;
42         number = newNumber;
43     }
44 }
45
46 class NamedAccount extends Account {
47     String fullName;
48
49     NamedAccount(String newFullName) {
50         balance = 0;
51         fullName = newFullName;
52     }
53 }
54
55 public class Main {
56     public static void main(String ... args) {
57         NumberedAccount numAccount = new NumberedAccount(891893133);
58         numAccount.deposit(100.0);
59         numAccount.withdraw(15.0);
60
61         NamedAccount namedAccount = new NamedAccount("Петър Петров");

```


62	namedAccount.credit(500.12);
63	namedAccount.debit(93.99);
64	}
65	}

Разпечатка 6. Наследяване на класове

Примерът на разпечатка 6 е опит да се създаде реалистичен пример за наследяване в контекста на банковите сметки. В нашата хипотетична банка имаме два вида сметки – такива, които са поименни и такива, които имат само номер. И с двата вида сметки обаче могат да се извършват еднакви действия – дебитиране, кредитиране, теглене и депозирание. Дебитирането и кредитирането представляват съответно премахване или добавяне към наличността. Тези операции се прилагат при теглене и депозирание, но и в други случаи, като например олихвяване на наличността или прилагане на банкови такси. Тегленето и депозирание на суми в нашата хипотетична банка са особено важни операции от гледна точка на информационната сигурност, поради което те се отразяват в специален журнал. Разбира се, тъй като нашият пример е учебен, записването в този журнал е представено чрез отпечатване на конзолата.

За да реализираме очертаната функционалност, ние сме създали три класа, които се възползват от механизмите на наследяването, за да елиминират повтарянето на еднакви методи. Първият от тях - класът **Account** – представлява сметка „по принцип“, т.е. общите неща между всички сметки, били те номерирани или поименни. Полето **balance** на този клас представлява наличността по сметката. Методите **debit** и **credit** извършват съответно дебитиране и кредитиране на сметката, а **withdraw** и **deposit** – съответно теглене и внасяне в сметката. Полето **balance** и четирите споменати метода са еднакви както за поименните, така и за номерираните сметки, което е точно и причината, поради която те са поставени в **Account**.

Класът **NumberedAccount** наследява класа **Account** и така „получава наготово“ полетата и методите на **Account**. Именно поради тази причина неговият конструктор може да зададе стойност на полето **balance**, както е

направено на ред 41 на разпечатката. По същата причина класът **NumberedAccount** има и методите **debit**, **credit**, **withdraw** и **deposit**, наличието на които е демонстрирано на редове 58 и 59. **NumberedAccount** обаче има и уникално поле, а именно полето **number**, което е дефинирано на ред 38 и инициализирано в конструктора няколко реда по-долу.

Абсолютно по същия начин класът **NamedAccount** наследява **Account** и поручава полето **balance**, заедно с методите **debit**, **credit**, **withdraw** и **deposit**, добавяйки към тях уникалното за класа поле **fullName**.

Можем да мислим за наследяването като процес, при който компилаторът на Java копира програмния код на всички методи от класа – родител към класа – наследник¹. В този смисъл компилаторът поема от програмиста задачата по дублирането на нужните методи, елиминирайки нуждата от преписване на един и същи код.

Така например в пример 6 компилаторът „прекопира“ за нас методите **withdraw**, **deposit**, **debit** и **credit** от **Account** към неговите наследници **NumberedAccount** и **NamedAccount**.

Наследяването обаче има и една много важна друга възможност, която го прави нещо повече от инструмент за копиране и увеличава неимоверно неговата полезност за програмистите. **Наследяването позволява т.нар. „предефиниране“ на методи, при които определени методи от класа-родител в процеса на копирането могат да се заменят с нови методи, предназначени специално за класа-наследник.**

Ние ще илюстрираме предефинирането на метод с допълнителен клас - **OverdraftAccount**, който ще добавим към програмата на разпечатка 8.6. Този клас ще представлява сметка с възможност за овърдрафт, т.е. временно изтегляне на по-големи суми, отколкото има налични по сметката.

¹ Читателите, които се притесняват за използването на паметта от Java, могат да бъдат спокойни. Копирането всъщност не се извършва в паметта, а за целта се използват таблици на виртуалните методи, които предотвратяват дублирането на програмен код.

Добавянето на овърдрафт променя начина по който се дебитират и кредитират сметките, което означава, че не можем просто да създадем клас - наследник на **Account**, който да получи идентични копия на методите **debit** и **credit**. Програмният код на тези методи трябва да се промени в **OverdraftAccount**. Програмният код на **withdraw** и **deposit** обаче няма нужда да се променя. Единствената модификация на тези методи в **OverdraftAccount** трябва да бъде такава, че да ги накараме да извикват новите „версии“ на **debit** и **credit**. Предефинирането прави точно това, копирайки нужните методи от **Account** към **OverdraftAccount**.

```
1  class OverdraftAccount extends Account {
2      double overdraftLimit;
3      double overdraft;
4
5      void credit(double amount) {
6          if (overdraft>0) {
7              if (overdraft>=amount) {
8                  overdraft = overdraft - amount;
9                  amount = 0;
10             } else {
11                 overdraft=0;
12                 amount = amount - overdraft;
13             }
14         }
15         balance = balance + amount;
16     }
17
18     boolean debit(double amount) {
19         if (amount>balance+overdraftLimit-overdraft) return false;
20         balance = balance - amount;
21         if (balance<0) {
22             overdraft = overdraft+(-balance);
23             balance = 0;
24         }
25         return true;
26     }
27
28     OverdraftAccount(double newOverdraftLimit) {
29         balance = 0;
30         overdraftLimit = newOverdraftLimit;
31         overdraft = 0;
32     }
33 }
34
35 public class Main {
36     public static void main(String ... args) {
37         OverdraftAccount overdraftAccount = new OverdraftAccount(1000);
38         overdraftAccount.deposit(100); //Deposit of 100.0
39         overdraftAccount.withdraw(500); //Successful withdrawal of 500.0
40         System.out.println(overdraftAccount.overdraft); // 400.0
```

41	}
42	}

Разпечатка 7. Предефиниране на методи

Класът **OverdraftAccount** наследява полето **balance** на **Account** и добавя още две полета – **overdraft** и **overdraftLimit**. Полето **overdraftLimit** съдържа максималната стойност на овърдрафта, който е позволен за дадената сметка, докато **overdraft** съдържа текущата стойност на овърдрафта, т.е. изтеглените суми извън наличните по сметката.

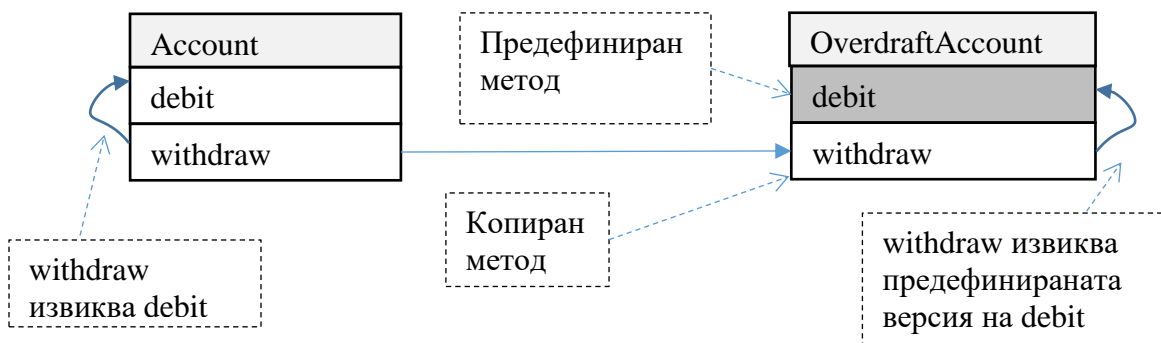
Както казахме по-горе, възможността да се теглят суми, по-големи от наличния баланс по сметката, изисква нови методи **debit** и **credit**.

Методът **debit** на **OverdraftAccount** проверява дали изтегляната сума надвишава максимално позволената, която е равна на сумата от баланса и максималния размер на овърдрафта, намалена с реализирания досега овърдрафт. Ако това е така, т.е. имаме надвишаване на овърдрафта, методът връща веднага **false** и не извършва дебитиране. В противен случай методът задава стойностите на полетата **overdraft** и **balance** така, че да отразяват направеното плащане.

Методът **credit** проверява дали има реализиран овърдрафт. Ако има такъв, той използва вноската за покриване на овърдрафта преди евентуално да добави оставащата сума към баланса по сметката.

Най-важното, което читателите трябва да видят в примера, е ефектът от предефинирането на методите. В метода `main` ние извикваме **deposit** и **withdraw**, които са наследени от **Account** без промени. Когато **deposit** и **withdraw** на свой ред викат **debit** и **credit** обаче, те извикват новите, предефинирани методи, предоставени от **OverdraftAccount** вместо старите такива, дефинирани в класа **Account**.

Позволяваме си да илюстрираме този много важен факт с диаграма за взаимодействието на **withdraw** и **debit**.



Фиг. 8. Предефиниране на методи

За да предефинираме даден метод, трябва да изпълним две условия:

Първо, новият метод в класа-наследник и оригиналният метод трябва да имат идентични сигнатури. С други думи новият метод трябва да има точно същото име като оригиналният метод, а списъкът му с аргументи трябва да отговаря по брой и тип на оригиналният метод.

Второ, ако оригиналният метод връща стойност от елементарен тип, предефинираният метод в класа-наследник трябва да връща стойност от същия тип. Ако оригиналният метод връща обект, предефинираният метод трябва да връща инстанция на същия клас или инстанция на субклас.

Програмата на следващата разпечатка илюстрира двете правила с няколко примера:

```

1 class Parent {
2     void drop(int test) { }
3     int compute(int quantity) { return 0; }
4     CharSequence reflect() { return null; }
5     CharSequence multiply(int a, double b, int c) { return null;};
6 }
7
8 class Child extends Parent {
9     int drop(int test) { return 0; } // Грешка при компилация
10    int compute(int quantity, int test) {return 0;}; //!!!
11    String reflect() {return null; }; // OK
12    CharSequence multiply(int p, double q, int r) {return null;}; //OK
13 }

```

Разпечатка 8. Предефиниране на методи

Разпечатка 8 съдържа два класа с примерни методи. Класът **Parent** е родител на класа **Child**. На ред 9 класът **Child** дефинира метод **drop**.

Въвеждането на този метод не води до предефиниране на метода **drop** в **Parent**, тъй като двата метода връщат различни по тип стойности. Новият метод връща **int**, докато старият метод връща **void**. При това положение Java ще се опита да третира двата метода като овърлоуднати, т.е. като варианти на един и същ метод, които се извикват с различни аргументи. За съжаление това няма да работи, тъй като и двата метода имат еднакви по тип аргументи. Методът **drop** на **Parent** взема един аргумент от тип **int**. Методът **drop** на **Child** прави точно същото. Дори имената на аргументите да бяха различни, техните типове са еднакви, което ще доведе до объркване кой от двата метода трябва да бъде извикан, когато в програмата например срещнем извикване от рода на **drop(5)**. Ето защо ред 9 не само не представлява пример за валидно предефиниране на метод, но и води до грешка при компилация.

Методът **compute**, дефиниран на ред 10 в **Child**, също не води до предефиниране на метода **compute** в **Parent**. И двата метода връщат **int**, но списъците с аргументи са различни. При това положение Java ще третира методите като овърлоуднати. Декларацията **compute(5)** например ще води до извикването на метода **compute**, дефиниран в **Parent** и наследен от **Child**, докато декларацията **compute(5,1)** ще води до извикването на метода, дефиниран от **Child**.

Методът **reflect** е пример за успешно предефиниране. Методът **reflect** в **Parent** и методът **reflect** в **Child** имат еднакъв списък с аргументи, който всъщност е празен. Методът **reflect** в **Child** обаче връща **String**, докато **reflect** в **Parent** връща **CharSequence**. Тъй като **CharSequence** е родител на **String**, това обаче е допустимо и удовлетворява второто условие, изложено по-горе. Поради тази причина предефинирането е успешно и методите на **Parent**, които са наследени от **Child** ще викат предефинираната версия без проблеми.

Читателите следва да обърнат внимание на факта, че наследените от **Parent** методи ще очакват **reflect** да върне обект от тип **CharSequence**, а вместо това ще получат обект от тип **String** от предефинираната версия на метода. Това обаче не

е проблем, тъй като **String** е наследник на **CharSequence** и следователно има всички методи и полета, които трябва да има и в **CharSequence**. Тоест **String** може да се използва и като **CharSequence** обект.

Това наблюдение, а именно че класовете-наследници могат да се използват навсякъде, където прогамата очаква клас-родител, всъщност е много сериозно и е свързано със следващата техника на ООП, която ще разгледаме в следващата точка.

Последното предефиниране на метода **multiply** също е успешно. Тук и двата метода връщат еднакви по тип стойности, а параметрите им са еднакви по тип, тоест първите параметри са от тип **int**, вторите параметри са от тип **double**, а третите – **int**. Независимо от различните вътрешни за метода имена, методът **multiply** в **Child** предефинира метода **multiply** в **Parent**.

На този етап вероятно е видно, че предефинирането на методи е податливо на прости грешки, свързани с неправилното изписване на сигнатурата на метода в класа-наследник. Долният пример илюстрира такава грешка:

```
1 package uk.co.lalev.javabook;
2
3 class Parent {
4     String getName() {
5         return "Аз съм класът-родител!";
6     }
7
8     void printName() {
9         System.out.println(getName());
10    }
11 }
12
13 class Child extends Parent {
14     String getname() { //!!!!
15         return "Аз съм класът-наследник";
16     }
17 }
18
19 public class Main {
20
21     public static void main(String[] args) {
22         Parent parent = new Parent();
23         Child child = new Child();
24         parent.printName(); // Аз съм класът-родител!
25         child.printName(); // Аз съм класът-родител!
26     }
27 }
```

В разпечатка 9 вероятно по погрешка на ред 14 програмистът този път е изписал името на метода само с малки букви. По този начин сигнатурата на метода **getname** сега се различава от сигнатурата на метода **getName** в **Parent** (ред 4). Това означава, че Java няма да третира метода **getname** в **Child** като предефиниращ метода **getName** в **Parent**. Като резултат **Child** ще има два метода - **getName** (наследен от **Parent**) и **getname**. Нещо повече, когато извикаме **child.printName()**, този метод ще извика оригиналния метод **getName**, наследен от **Parent**. Както е видно от разпечатката, това ще доведе до извеждането на “Аз съм класът-родител!” на ред 25.

Проблемът идва от това, че компилаторът на Java няма как да отгатне намеренията на програмиста. Дали става въпрос за грешка в изписване на метода и неговите параметри, или програмистът е имал предвид да създаде нов метод, който не предефинира никой от наследените методи?

За да избегне подобни дилеми и трудни за откриване елементарни грешки, Java позволява предефинираните методи да бъдат *анотирани*, като пред тях бъде поставена анотацията **@Override**.

Анотацията **@Override** казва на компилатора, че програмистът има намерение да предефинира метод. Ако се окаже че сигнатурата на така анотирания метод не съвпада с никой от методите на класа-родител, компилаторът ще знае, че става въпрос за грешка при изписване на името или параметрите на метода и ще изведе предупредително съобщение, спирайки компилацията на програмата.

Долният програмен сегмент демонстрира как точно се анотират предефинираните методи:


```

6  class Parent {
7      String getName() {
8          return "Аз съм класът-родител!";
9      }
10
11     String getFoo() {
12         return "foo";
13     }
14 }
15
16 class Child extends Parent {
17     @Override                                //Грешка при компилация
18     String getname() {
19         return "Аз съм класът-наследник";
20     }
21
22     @Override
23     String getFoo() {
24         return "bar";
25     }
26 }

```

Разпечатка 10. Анотация `@Override` за обозначаване на предефинирани методи

Този път **Child** предефинира и двата метода, наследени от **Parent**. На ред 17 програмистът започва декларация на такъв предефиниран метод като поставя анотацията **@Override**. Сигнатурата на метода на ред 18 обаче не съвпада с никоя от двете сигнатури на класа **Parent**. Ето защо вместо да продължи, компилаторът ще спре и ще изведе грешка. Тук читателите следва да забележат разликата в поведението на компилатора, породено от използването на **@Override**. В предния пример тя липсваше, поради което грешката в името на метода не бе открита и компилаторът просто реши, че това е нов, напълно различен метод.

Редове 22-25 демонстрират, че когато сигнатурата на предефинирания метод е изписана правилно, **@Override** не спира компилацията и кодът се компилира коректно. Тук е моментът да споменем, че дори програмистът да е сигурен в това, че изписва сигнатурата на даден предефиниран метод коректно, той *винаги* трябва да използва анотацията **@Override**. Обратното е проява на изключително лош стил.

Завършваме точката с още няколко важни факта за наследяването:

1. В Java не е позволено наследяването на множество класове

Всеки клас в Java може да наследява един-единствен клас.

2. Всички класове в Java наследяват класа **Object**

Ако програмистът не маркира, че даден клас наследява друг, компилаторът по подразбиране ще направи класа наследник на специалния клас **Object**. Така всички класове в Java са наследници на **Object** директно или чрез друг клас/класове над тях в обектната йерархия.

3. „Цикличното“ наследяване не е позволено в Java

Не е възможно даден клас да наследява сам себе си директно или чрез поредица от междинни класове. Долният програмен сегмент илюстрира втория случай:

6	class First extends Third {}	//Грешка при компилация
7		
8	class Second extends First {}	//Грешка при компилация
9		
10	class Third extends Second {}	//Грешка при компилация

Разпечатка 11. „Цикличното“ наследяване не е позволено

В показания на разпечатката пример програмистът задава че клас **First** наследява **Thrid**, който от своя страна наследява **Second** и **First**.

3. Полиморфизъм

Започваме дискусията на третата важна техника на ООП там, където приключихме дискусията на втората. Коментирайки предефинирането на методи, ние уточнихме, че когато оригиналният метод връща инстанции на даден клас, предефинираният метод може да връща инстанции на всеки клас-наследник или субкласове на същия този клас. Причината за това беше, че **обект от типа на класа-наследник притежава всички методи и полета на класа-родител, независимо дали методите са директно наследени или предефинирани. По-прецизно казано, класът наследник може да се държи различно от класа-родител, благодарение на предефинираните методи, но самите методи се**

извикват идентично. Това означава, че навсякъде в програмата, където се изисква инстанция на класа-родител, ние можем да предоставим инстанция на всеки от класовете-наследници (или техни субкласове). Това „подменяне“ не изисква пренаписване на какъвто и да било код и дори може да се случва по време на изпълнението на програмата. Ние наричаме това „полиморфизъм“.

Използван правилно, полиморфизмът не само не води до грешки, но е и изключително полезен за опростяване структурата на програмите. Следващите разпечатки разширяват нашите примери от точката за наследяване като демонстрират използването на полиморфизъм.

Ще започнем като демонстрираме полиморфизъм при променливите, който се изразява в това, че **променливите от типа на даден клас могат да сочат към инстанции на всеки от субкласовете на този клас.**

```
51 public class Main {
52     public static void main(String ... args) {
53         Account test = new NumberedAccount(8934);
54         test = new NamedAccount("Елена Николова");
55
56         Account accounts[] = new Account[6];
57         accounts[0] = new NumberedAccount(7910);
58         accounts[1] = new OverdraftAccount(2200);
59         accounts[2] = new NamedAccount("Боян Григоров");
60         accounts[3] = new NumberedAccount(341);
61         //accounts[4] = new Object();    // Грешка при компилация
62         //accounts[5] = new Random();    // Грешка при компилация
63
64         for (int i=0; i<3; i++) {
65             accounts[i].credit(10.0);
66         }
67     }
68 }
```

Разпечатка 12. Демонстрация на полиморфизъм

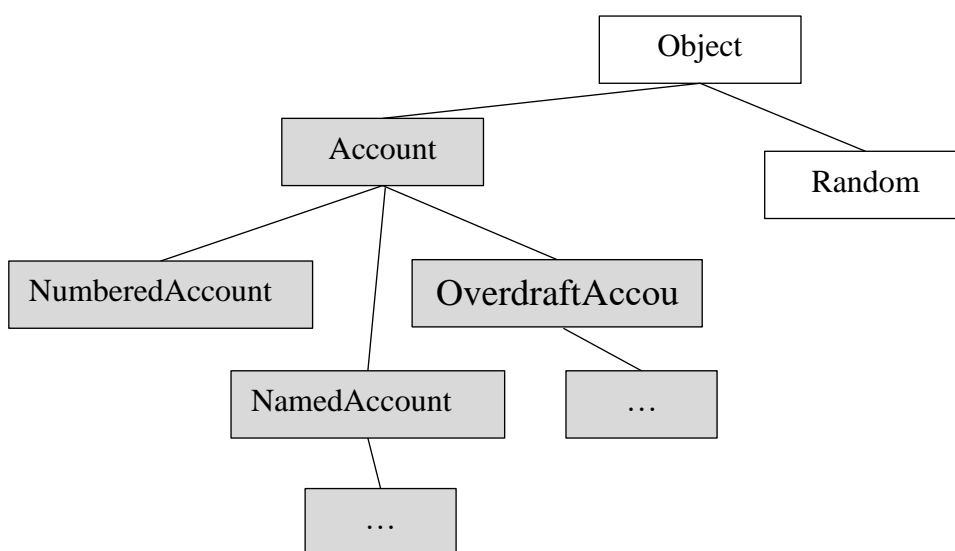
Програмата от разпечатка 12 многократно съхранява променливи от типове-наследници на **Account** в променливи от тип **Account**. Така например, на редове 53 и 54 ние последователно присвояваме на променливата **test** от тип **Account**, обекти от тип **NumberedAccount** и **NamedAccount**.

Дори по-интересни са присвояванията на редове 57-60, които демонстрират, че полиморфизмът работи по същия начин за елементите на масивите. Всеки от елементите на масив от типа на даден суперклас може да съдържа инстанции на субклас. Така например на ред 60 ние присвояваме на **accounts[3]** елемент от тип **NumberedAccount**, въпреки че масивът е от тип **Account**.

Важен факт за полиморфизма, върху който не можем да наблегнем достатъчно, е това че полиморфизмът работи само „надолу“ по обектната йерархия. Двете грешки при компиляция имат за цел да демонстрират точно това.

Грешката на ред 58 се дължи на това, че **Object** не е клас-наследник на **Account**. Всъщност е точно обратното, **Object** е суперклас на всички класове в Java, следователно **Object** е такъв и за **Account**. Но в променлива от тип **Account** ние *не можем* да поместваме класове-родители, а само класове-наследници.

Класът **Random** (най-вероятно от **java.util**) също е наследник на **Object**, но не е нито родител, нито наследник на **Account**. Това означава отново, че не можем да поместим референция към **Random** в променлива от тип **Account** (вж. фиг. 8.3)



Фигура 3. Типове обекти, които могат да се поставят в променливи от тип **Account**.
На диаграмата тези обекти са оцветени в сиво.

Последните редове на примера започват да демонстрират за какво точно е полезен полиморфизма. На редове 64-66 ние имаме цикъл, който обхожда правилно инициализираните елементи на нашия масив **accounts** и извиква техните методи **debit**. Ако няхаме полиморфизъм, вместо един цикъл, трябваше да напишем два, тъй като щяхме да имаме два отделни масива – един за инстанции на **NumberedAccount** и един за инстанции на **NamedAccount**. Читателите могат лесно да си представят какви затруднения бихме изпитвали в с нарастване на броя на класовете-наследници на **Account**, ако не разполагахме с механизмите на полиморфизма.

Полиморфизмът има и едно важно ограничение. Ако в променлива от даден клас поставим инстанция на клас-наследник, ние не можем чрез тази променлива да използваме новите методи и полета, дефинирани в класа-наследник. Правилото важи за обектната йерархия по принцип. Ако нашият хипотетичен клас е А, който е родител на В, а той на свой ред е родител на С, чрез променлива от тип А, която сочи обект от тип С, не можем да използваме нито новите методи и полета, дефинирани от В, нито новите методи и полета, дефинирани от С.

Често давана аналогия, която пояснява току-що демонстрирания проблем, оприличава полиморфизма на управлението на моторно превозно средство. Шофьор на лек автомобил (нашата променлива от типа на класа-родител), който за пръв път се опитва да шофира тежък камион например (обект от тип клас надолу по йерархията, който има много специфична функционалност), би могъл да разпознае и използва универсалните и общи инструменти за управление, като кормило, педали за газ и спирачки, скоростен лост (методи на класа родител), независимо от това че тези инструменти могат реално да работят по различен начин спрямо леката кола (предефинирани методи). Но без допълнителни знания, такъв шофьор не би могъл да използва специфичните за камиона инструменти като радио, контрол на трансмисията, осите и т.н. (специфични за класа надолу по йерархията методи).

Долният пример представлява конкретна илюстрация на току-що казаното:

```
1 package uk.co.lalev.javabook;
2
3 class Parent {
4     int parentsField;
5     void parentsMethod() {
6         System.out.println("parentsMethod from Parent");
7     }
8 }
9
10 class Child extends Parent {
11     int childsField;
12     @Override
13     void parentsMethod() {
14         System.out.println("parentsMethod from Child");
15     }
16     void childsMethod() {
17         System.out.println("childsMethod from Child");
18     }
19 }
20
21 public class Main {
22     public static void main(String[] args) {
23         Parent p = new Child();
24         System.out.println(p.parentsField);    // OK -> 0
25         p.parentsMethod();                    // OK -> parentsMethod from Child
26         //p.childsMethod();                  // Грешка при компиляция
27     }
28 }
```

Разпечатка 13. Ограничения на полиморфизма

Класовете **Parent** и **Child** на разпечатката са в отношение родител-наследник. Класът **Child** предефинира **parentsMethod** и добавя нов метод – **childsMethod**. Както можем да видим от редове 23-26, когато поставим обект от тип **Child** в променлива, която е от тип **Parent**, ние нямаме проблем с достъпа до полетата или методите на **Parent** (ред 24). Нещо повече, ако **Child** предефинира даден метод на **Parent**, когато използваме променлива от тип **Parent** за достъп до **Child** обект, ние всъщност ще извикаме *предефинирания метод* в **Child** (ред 25).

Единственият проблем възниква, когато опитаме да достъпим методи или полета, които съществуват само в **Child**. Това води до грешка при компиляция, както е демонстрирано на ред 26.

Добрата новина за споменатото ограничение е това, че то е обратимо. Класовете в Java могат да се преобразуват подобно на елементарните типове,

дискутирани в глава 2. Ние ще разгледаме преобразуването на типове подробно в следващите глави.

Полиморфизмът работи и при предаване на параметри, което го прави още по полезен. Долната програма илюстрира как полиморфизмът спестява нуждата от множество овърлоуднати методи. За целта тя добавя метод **transferTo** към класа **Account**.

```
... // Във Account
32 void transferTo(Account account, double amount) {
33     if (debit(amount)) {
34         account.credit(amount);
35     }
36 }
...
51 public class Main {
52     public static void main(String[] args) {
53         NumberedAccount n = new NumberedAccount(58911);
54         NamedAccount i = new NamedAccount("Петър Петров");
55         OverdraftAccount o = new OverdraftAccount(1000.0);
56         o.transferTo(i, 100.0);
57         System.out.println("Overdraft: "+o.overdraft); // Overdraft:100.0
58         o.transferTo(n, 100.0);
59         System.out.println(("Overdraft: "+o.overdraft); // Overdraft:200.0
60         i.transferTo(n, 50.0);
61         System.out.println("Balance:"+i.balance); // Balance:50.0
62         System.out.println("Balance:"+n.balance); // Balance:150.0
63     }
64 }
```

Разпечатка 14. Демонстрация на полиморфизъм

Методът **transferTo** взема като аргумент друга сметка, към която ще бъдат прехвърлени средства, както и сумата, която ще бъде прехвърлена. За да извърши трансфера, **transferTo** вика метода **debit** на текущата сметка, след което извиква **credit** на сметката, подадена като аргумента **account**. Методът **debit** от своя страна ще върне **true**, ако в текущата сметка има достатъчно средства, за да се извърши трансфера. Ние използваме този факт, за да прекратим трансфери, за които няма средства.

Забележителното в примера идва от начина, по който използваме **transferTo** в метода **main**. На ред 56 викаме **o.transferTo**, като подаваме за параметър променливата **i**, която е от тип **NamedAccount**. Читателите веднага

могат да погледнат към ред 32 на разпечатката, за да забележат, че дефиницията на **transferTo** изисква като параметър на тази позиция обект от тип **Account**.

Благодарение на полиморфизма и тъй като **NamedAccount** е клас-наследник на **Account**, ние можем да направим това без да предизвикаме грешка при компилация. Тъй като **NamedAccount** има всички методи на **Account**, кодът и програмната логика на метода **transferTo** ще работят без проблеми. Така например когато на 34 ред извикваме **account.credit** това ще извика метода **credit** на **NamedAccount**, който е директно наследен от **Account**.

Следващите редове са още демонстрации в подобен стил. Така например на ред 58 подаваме **NumberedAccount** вместо **Account**, а на ред 60 подаваме отново **NamedAccount** вместо **Account**.

Най-хубавото на метода **transferTo** от гледна точка на избягване повторното писане на подобен код, е това, че ако в даден момент ние програмираме нов клас, наследник на **Account**, който представлява друг вид сметка, **transferTo** ще може да взема като аргумент и този клас без каквито и да е промени, тъй като този бъдещ клас ще има всички методи, наследени от **Account** и необходими на **transferTo**, за да извърши функциите си.

4. Скриване

Скриването може да бъде разглеждано като мощна надстройка на голямата идея, залегнала зад инкапсулацията, а именно това, че програмистът-ползвател на класа не трябва да знае детайли за вътрешната организация на този клас, а само за начините, по които той се използва.

В примерите от предходната точка ние видяхме, че Java по принцип дава на програмистите възможности за достъп до всеки метод и всяко поле, независимо дали става въпрос за текущия или друг клас. Ако разполагахме само с механизма на инкапсулацията, това щеше да означава, че всякакви ограничения за това как може да се използва даден клас (като например коментарите за

добрите практики точно в края на предходната точка) щяха да имат само препоръчителен характер.

Скриването позволява на създателите на класове да налагат много по-стриктни правила за това кои полета и методи могат да се използват от програмистите-ползватели на тези класове.

Скриването се извършва на няколко нива, като най-важните от тях са нивото на класовете и нивото на пакетите. Скриването се осъществява с помощта на ключови думи – т.нар. „**модификатори за достъп**“. Повечето от тези модификатори бяха част от множество примерни програми в предходните глави и ние упорито избягвахме да ги подложим на дискусия досега.

4.1. Скриване на ниво класове

В дефиницията на всяко поле и метод могат да се поставят три различни модификатора за достъп. Също така е възможно пред полето или метода да не се постави никакъв модификатор. Това дава четири варианта за ограничаване на достъпа.

Модификаторът **public**, поставен пред дефиницията на поле или метод, означава че на практика липсват всякакви ограничения на достъпа.

Модификаторът **private** от друга страна означава, че този метод или поле няма да може да се използва *извън* дадения клас.

Модификаторът **protected** означава, че съответното поле или метод могат да се достъпват от същия клас, от всички класове, които се намират в същия пакет, или от класове, които не са в същия пакет, но са наследници на дадения клас.

Липсата на модификатор обозначава ситуацията по подразбиране, която позволява достъп до съответното поле или метод само от същия клас или от класове в същия пакет.

Долният пример демонстрира използването на **private** и **public** модификатори.

```
1 package uk.co.lalev.javabook;
2
3 class Person {
4     public String name;
5     public String middleName;
6     public String surname;
7     private int age;
8 }
9
10 public class Main {
11     public static void main(String[] args) {
12         Person person = new Person();
13         person.name="Нина";
14         person.middleName="Лазарова";
15         person.surname="Лалева";
16         //person.age = 50; //Грешка при компилация
17     }
18 }
```

Разпечатка 15. Използване на модификатори **public** и **private**

В примера полетата **name**, **middleName** и **surname** са отбелязани с модификатор **public**, което означава, че те могат да бъдат достъпвани от други класове. Това е причината, поради което методът **main** от класа **Main** може да осъществи достъп до тези полета без проблеми на редове 13-15.

Модификаторът **private** от друга страна забранява използването на полето **age** от методи, разположени в други класове, без значение къде са разположени тези класове и какво е отношението им към класа **Person**. Тъй като методът **main** е разположен извън класа **Person**, той не може да достъпи полето **age**, което води до грешка по време на компилация.

Читателите, които тепърва се сблъскват с модификаторите за достъп и скриването, най-вероятно лесно ще си представят защо можем да декларираме даден *метод* **private**. Нуждата от правилна организация и избягване повторното писане на код водят до това, че в класовете много често имаме различни помощни методи, които съществуват само за да подпомогнат работата на основните методи на класа. Ние вече демонстрирахме подобни методи и в примерите от настоящата глава. Когато програмистът прецени, че няма причини (и/или полза) такива

методи да се извикват директно от външни класове, той обикновено маркира тези методи **private**. Скриването всъщност улеснява ползвателите на класа, тъй като то ясно маркира кои методи са вътрешни за класа (**private**) и кои методи всъщност са начините, по които външни класове следва да използват класа (**public**). Нещо повече, скритите методи на даден клас могат да бъдат променяни свободно без да се налагат никакви промени по субкласовете надолу по обектната йерархия.

Малко по-объркващ за начинаещите е демонстрираният случай на полета, маркирани **private**. Защо бихме искали да ограничим достъпа на външни класове до дадено поле и как тогава ние можем да поставим стойност в него? Най-често причината за подобно решение се корени в нуждата на програмиста да контролира смислеността на входните данни.

Връщайки се към пример 14 ние виждаме, че полето **age** е от тип **int**. Нищо не пречи в него да поставим например **-60**, въпреки че отрицателните стойности нямат смисъл в конкретния контекст. По същия начин нищо не пречи да поставим празен текст за полето **name**, въпреки че няма хора без собствени имена. Въпреки че подобни грешки изглеждат наивни, когато кодът стане сложен и се сподели между множество програмисти, става много лесно един програмист да направи грешка. Ситуации с грешни входни данни, към които се прилага иначе коректно описана логика на реализираната задача, се срещат толкова често, че си имат неофициално име - „*garbage in – garbage out*” („*влиза боклук – излиза боклук*“). Оказва се, че скриването предлага идеално решение за избягването им. Конкретният начин, по който се прави това, е толкова често срещан в обектно-ориентираните програми, че читателят трябва задължително да го проумее и запомни.

```

1 package uk.co.lalev.javabook;
2
3 class Person {
4     private String name;
5     private int age;
6
7     public String getName() { return name; }
8
9     public void setName(String newName) {
10         if (newName.isEmpty()) //Проверява за празен симв. низ
11             throw new RuntimeException(); //Сигнализира за грешка
12         name = newName;
13     }
14
15     public int getAge() { return age; }
16
17     public void setAge(int newAge) {
18         if (newAge<0) throw new RuntimeException(); //Сигнализира за грешка
19         age = newAge;
20     }
21 }
22
23 public class Main {
24     public static void main(String[] args) {
25         Person person = new Person();
26         person.setAge(20);
27         System.out.println(person.getAge());
28         person.setAge(-20); //Грешка!
29         person.setName(""); //Грешка!
30     }
31 }

```

Разпечатка 16. Използване на гетери и сетери

Кодът на разпечатка 16 представлява модификация на кода от пример 15. От съображения за краткост на примера този път класът **Person** има само две полета – **name** и **age**. Полето **name** съдържа име, а **age** съдържа възраст на лицето. И двете полета са маркирани **private**, което означава че външните класове не могат да боравят *директно* с тях. За да позволи на външни класове все пак да задават и изчитат подават стойност, програмистът предоставя по два метода на поле.

За полето **name**, тези методи се казват **setName** и **getName**. И двата метода са публични, което означава, че за разлика от самото поле, те *могат* да се извикват от външни класове! Предназначението на **setName** е да задава нова стойност на полето **name**. Методът **getName** пък връща стойността на полето **name**.

Благодарение на двата метода, макар и индиректно, външните класове могат да променят и използват стойността на полето `name`. Това, което се печели чрез такъв индиректен достъп до стойността на полето `name`, е възможността в метода `setName` да се поставят проверки за смислеността на новата стойност за полето `name`.

Точно по същия начин работят методите `getAge` и `setAge`. `getAge` позволява изчитане на стойността, поставена в **private** полето `age`, но отново същинското подобрение е възможността да проверим смислеността на новата стойност за `age` само веднъж, още когато тази нова стойност се поставя в полето чрез метода `setAge`. При отрицателна стойност на параметъра `newAge`, методът `setAge` ще генерира изключение, което ще спре изпълнението на програмата.

Програмистите на Java наричат методите като `setAge` и `getAge` съответно „сетери“ (“setters”) и „гетери“ (“getters”). Тези методи имат и „теоретични“ имена, които са съответно „мутатори“ (“mutators”) и „аксесори“ (“accessor”). Читателите следва да забележат, че в много ситуации е полезно да се пишат гетери и сетери, дори да нямат нужда от конкретна програмна логика за проверка на смисленост. Това позволява по-лесното вмъкване на такава логика по-късно, когато даденият клас ще има вече написани класове-ползватели.

Читателите, които внимателно са прочели главата до момента веднага ще направят паралел между гетерите и сетерите и конструкторите. Общото между двете, разбира се, е това, че те извършват валидация на данните, които „влизат“ в обекта. Разликата между гетерите и сетерите и конструкторите обаче е в това кога се извършва тази валидация. Конструкторите валидират началните данни, докато гетерите и сетерите валидират последващите изменения в състоянието на обекта, зададени „отвън“, т.е. от външни класове и методи.

Използваме момента да отбележим, че докато обектите, които променят състоянието си под въздействие на външни методи чрез гетери и сетери са често срещани, програмистите понякога предпочитат обектите да не се изменят

по време на живота си. Подобни обекти се наричат „неизменчиви“ или „**immutable**“ и са една още една илюстрация за гениалността на инкапсулацията.

Неизменчивите обекти получават начална стойност през конструктора и имат само **private** полета, съчетани само с гетери. По този начин външните методи могат да четат, но не и да променят състоянието на обектите, веднъж щом тези обекти са създадени. Когато искаме да редактираме състоянието на такива обекти ние просто копираме данните от стария в нов обект като в процеса на копиране нанасяме желаните редакции.

Неизменчивите обекти опростяват многонишковото програмиране, което е и едно от най-големите предизвикателства към знанията и уменията на програмистите на Java и много други езици. Ето защо читатели, желаещи да усвоят допълнителни знания могат да потърсят информация в Интернет за проектирането на подобни обекти.

Приключваме точката с кратка дискусия на другите два модификатора. Тези модификатори типично се използват когато програмистът на даден клас иска да даде повече права на програмистите, които ще програмират класове-наследници или други тясно-свързани класове, в сравнение с програмистите, които просто ще използват дадения клас.

Използването на достъп по подразбиране е демонстрирано в долния пример:

Test.java	
1	package uk.co.lalev.javabook;
2	
3	public class Test {
4	public int pub;
5	private int priv;
6	int def; //Режим на достъп по подразбиране
7	}
Main.java	

1	package uk.co.lalev.javabook;
2	
3	public class Main {
4	public static void main(String[] args) {
5	Test t = new Test();
6	t.pub = 5; //OK. Достъп до public поле
7	t.priv = 6; //Грешка! Опит за достъп до private поле
8	t.def = 7; //OK. Поле с режим на достъп по подразбиране, достъп
9	//от клас в същия пакет.
10	}
11	}
Remote.java	
1	package uk.co.lalev.javabook.other;
2	
3	import uk.co.lalev.javabook.Test;
4	
5	public class Remote {
6	public void test() {
7	Test t = new Test();
8	t.pub = 5; //OK. Достъп до public поле.
9	t.priv = 6; //Грешка! Опит за достъп до private поле
10	t.def = 7; //Грешка! Поле с режим на достъп по подразбиране. Достъп
11	// от клас извън пакета.
12	}
13	}

Разпечатка 17. Демонстрация на режим на достъп по подразбиране

Програмата на разпечатка 17 се състои от три класа, като класът **Remote** е дефиниран в различен пакет от останалите два. Класът **test** има три полета с модификатори съответно **public**, **private** и „по-подразбиране“ (тоест липсващ модификатор за достъп).

На редове 6 и 7 в **Main.java** ние опитваме поред достъп до **public** и **private** полето на класа **Test**. Тъй като **public** на практика разрешава достъп безусловно и отвсякъде, ред 6 се компилира успешно. Ред 7 обаче генерира грешка при компилация, тъй като **private** полетата са достъпни само за методите на собствения им клас. Ред 8 демонстрира свойствата на режима на достъп по подразбиране. Тъй като **Main** и **Test** са в един пакет, **Main** може да достъпи без проблем полето и редът се компилира. Не такъв обаче е случаят с ред 10 в **Remote.java**. Там се извършва напълно аналогичен опит за достъп, но този път **Remote** е в друг пакет спрямо **Test**. Резултатът е грешка при компилация на програмата.

Последният модификатор – protected – позволява дори по-свободен достъп до маркираните с него полета и методи. В допълнение на достъпа от други класове в същия пакет, **protected** методите и полетата могат да бъдат достъпвани и от класовете наследници.

Test.java	
1	package uk.co.lalev.javabook;
2	
3	public class Test {
4	public int pub;
5	protected int prot;
6	int def; //Режим на достъп по подразбиране
7	}
Remote.java	
1	package uk.co.lalev.javabook.other;
2	
3	import uk.co.lalev.javabook.Test;
4	
5	public class Remote extends Test{
6	public void test() {
7	pub = 5; //ОК. Достъп до public поле.
8	prot = 6; //ОК. Клас-наследник извън пакета
9	def = 7; //Грешка! Поле с режим на достъп по подразбиране. Достъп
10	// от клас извън пакета.
11	}
12	}
13	

Разпечатка 18. Демонстрация на **protected** модификатори за достъп

Програмата от разпечатката демонстрира че **protected** е по-малко рестриктивен от достъпа по подразбиране. Класът-наследник **Remote** може да осъществи достъп до полето **prot** на **Test** (което има **protected** достъп). В същото време същият клас няма достъп до полето **def** (**default** достъп) на **Test**, тъй като се намира в различен пакет от **Test**.

4.2. Скриване на ниво пакети

Читателите без съмнение са забелязали в множеството примери от тази и предходната глава, че ние понякога използваме модификатора за достъп **public** и в дефиницията на класове, а не само при методи и полета.

Това е така, тъй като класовете също имат две нива на видимост (или респективно достъп).

Клас, който е деклариран с помощта на модификатора **public**, е видим и достъпен извън пакета в който е дефиниран. Такъв клас може да се наследява и/или използва от класове и методи, които се намират в други пакети.

Класовете, които нямат такъв модификатор, са достъпни само вътре в пакета, в който са дефинирани. Тоест, само класове от същия пакет могат ги разширяват или използват като типове на променливи.

Програмистите на Java не декларираят като **public** само определени обслужващи класове, които съществуват само за да позволят на основните класове да извършват своята дейност.

Java има важно изискване за класовете, маркирани като **public**. Тези класове трябва да са декларирани в отделен файл с разширение `.java` и име, което *дословно* отговаря на името на класа. Това означава, че не е възможно да се декларира два **public** класа в един файл.

Java не предвижда такива ограничения по отношение на класовете, които не са **public**. Тези класове могат да се намират както във файл със същото име, така и във файл с произволно име. Много често такива класове се поставят във файла на главния клас, който те подпомагат.

Читателите сигурно са забелязали, че в много от нашите примери, с цел по-голяма компактност на разпечатките, ние декларирахме само един от класовете като **public**, независимо от това, че останалите класове са толкова тясно свързани с него. По този начин избегнахме тромавата нотация, която произтича от нуждата да уточняваме в кой файл би следвало да се намират различните класове.

В реалния свят авторът се придържа към практиката всеки клас да се намира в отделен файл с неговото име и разширение `.java` и до момента не се е сблъскал със сериозни проблеми, породени от този избор.

Глава 9. Разширено обектно-ориентирано програмиране с Java¹

В настоящата глава ние ще продължим да представяме механизмите на обектно-ориентираното програмиране (ООП), насочвайки вниманието на читателите от общите принципи към специфичните за Java механизми.

1. Разширена инкапсулация

1.1. Статични методи и полета

В предходната глава, посветена на въведението в ООП, ние многократно наблегнахме на няколко твърдения, които бяха само донякъде верни. Там казахме че не класовете, а техните инстанции (наричани още „обекти“) съдържат реалните данни, обработвани от програмата. По-конкретно, ние уточнихме, че класът задава вида, имената и броя на полетата, но данните в полетата са различни за отделните обекти. Ние често мислим за тези полета като за променливи, принадлежащи на конкретния обект.

Ние също така споменахме, че методите на даден клас винаги се извикват чрез инстанция на този клас. С други думи при извикването на който и да било метод винаги е ясен обектът, с който методът е асоцииран, и чиито полета методът евентуално може да промени.

За съжаление тези две твърдения не отразяват някои изключения, които игнорирахме в предните глави в името на по-достъпното и добре-структурирано изложение.

Първото изключение е това, че някои полета могат да се числят към самия клас, а не към неговите инстанции. Такива полета се споделят от всички инстанции на класа и са понякога необходими, за да могат отделните инстанции

¹ Темата е разработена от гл. ас. д-р Ангелин Лалев

да обменят информация помежду си. Ние наричаме такива полета „*статични*“ (също неформално „*полета на класа*“). В програмния код ние обозначаваме статичните полета с ключовата дума **static**.

Важно е да се помни, че статичните полета са по-скоро изключение, отколкото правило. Тяхната твърде честа поява в една ООП програма може да подсказва лош стил на програмиране и непознаване на някои важни ООП шаблони за дизайн.

Следващият пример демонстрира използването на статични полета в клас:

```
1  class StaticTest {
2      public static int inventarenNomer;
3      static public double edinichnaCena;
4
5      public void nonStaticMethod() {
6          edinichnaCena = 5;
7      }
8  }
9
10 public class Main {
11     public static void main(String[] args) {
12         StaticTest.edinichnaCena = 15.30;
13         StaticTest.inventarenNomer = 23;
14
15         StaticTest t = new StaticTest();
16         System.out.println(StaticTest.edinichnaCena); // 15.30
17         System.out.println(StaticTest.inventarenNomer); // 23
18
19         StaticTest s = new StaticTest();
20         System.out.println(s.edinichnaCena); // 15.30
21         System.out.println(s.inventarenNomer); // 23
22
23         StaticTest.edinichnaCena = 0.0;
24         t.inventarenNomer = 0;
25         System.out.println(t.edinichnaCena); // 0.00
26         System.out.println(s.inventarenNomer); // 0
27         System.out.println(StaticTest.edinichnaCena); // 0.00
28     }
29 }
```

Разпечатка 9.1. Демонстрация на статични променливи

Читателите следва да забележат как е използвана ключовата дума **static** на редове 2 и 3 в разпечатката. Мястото на **static** задължително трябва да бъде *преди* типа на променливата. Няма други изисквания за позицията на **static** - тя може да бъде поставяна на произволно място спрямо другите модификатори, които стоят преди типа на променливата. Сред програмистите е широко разпространена

практиката **static** да се поставя *след* модификаторите за достъп, както е направено на ред 2.

Останалата част от демонстрацията има за цел да покаже, че статичните променливи се държат като част от самия клас, независимо от начина, по който се осъществява достъп до тях.

Ред 6 показва, че методите на класа могат да работят със статичните полета точно по същия начин, както и с останалите полета. За да се осъществи достъп до статично поле от метод в същия клас, е достатъчно в метода да се посочи само името на полето.

Редове 12 и 13 демонстрират, че ние можем да работим със статичните полета на един клас дори преди да имаме създадени обекти от този клас. От тези редове също е видно как един клас може да осъществи достъп до статичните полета на друг клас. Тъй като тези полета не са асоциирани с конкретна инстанция, а с целия клас, за префикс може да се използва името на класа.

Така например на ред 12 от разпечатката полето **edinichnaCena** е посочено като **StaticTest.edinichnaCena**. Разбира се, необходимостта от префикс възниква, тъй като **StaticTest** е външен за класа **Main**.

Останалите редове на програмата демонстрират в по-големи детайли вече описаното основно свойство на статичните полета — всяко такова поле принадлежи на самия клас и се споделя от обектите на този клас.

Редове 15 до 27 показват, че за достъп до статичните полета може да се ползва и името на променлива, сочеща към вече създаден обект на класа. Така например вместо **StaticTest.edinichnaCena**, както е написано на ред 12, ред 20 указва същото поле като **t.edinichnaCena**. Ред 25 отново указва същото поле като **s.edinichnaCena**.

Използването на конкретна инстанция за осъществяването на достъп до статични полета, макар и възможно, е лош стил на програмиране. Когато се налага използването на статично поле на друг клас, най-правилният префикс е името на самия клас, както е направено на редове 12, 23 и 27.

Статичните променливи се допълват от *статични методи*. Тези методи могат да се извикват по същия начин като статичните променливи – чрез самия клас вместо чрез обекти от същия клас.

Статичните методи на свой ред не могат да извикват не-статични методи и не могат да осъществяват достъп до не-статични полета на класа, без да посочат променлива, сочеща към обект от типа на класа. За сметка на това статичните методи могат да използват директно без префикс статичните променливи на класа. По същия начин те могат да викат директно други статични методи.

Работата със статични методи е демонстрирана в следващия пример:

```
1 public class Main {
2     private int test;
3     private static int test2;
4
5     public void doTest() {
6         System.out.println("Стойността на test е: "+test);
7         System.out.println("Стойността на test2 е: "+test2);
8     }
9
10    public static void doTest2() {
11        // System.out.println("Стойността на test е: "+test); // Грешка
12        System.out.println("Стойността на test2 е: "+test2);
13    }
14
15    public static void main(String[] args) {
16        //doTest(); // Грешка при компилация
17        //test = 5; // Грешка при компилация
18        doTest2(); // OK
19        test2 = 4; // OK
20        Main m = new Main();
21        m.test = 5; // OK
22        m.doTest(); // OK
23    }
24 }
```

Разпечатка 9.2. Демонстрация на статични методи

Програмата на разпечатка 9.2 има един не-статичен и един статичен метод (съответно **doTest** и **doTest2**), както и по една не-статична и статична променлива (**test** и **test2**). Интересното е, че сега, за пръв път в нашите примери, тези полета и методи са декларирани в класа, който съдържа входната точка на програмата – методът **main**.

Най-сетне можем да обърнем внимание на това, че входната точка на програмата – методът **main**, който беше част от всеки един наш пример досега - винаги е статичен метод. По този начин виртуалната машина може да го стартира без да трябва да създаде инстанция на класа. Това на свой ред означава, че не е задължително входната точка на програмата да бъде обособена в собствен клас, както правихме в примерите до момента, а може да бъде добавена към други класове, които съдържат функционалността на програмата.

В пример 9.2 правим точно това и цялата програма, включително методи, полета и входната точка, се съдържа в само един клас.

Както споменахме по-горе, статичните методи не могат да извикват не-статични методи директно, тъй като при статичните методи нямаме текущ обект.

Това е демонстрирано на ред 16 в примера, където статичният метод **main** прави опит да извика не-статичния метод **doTest** директно, което предизвиква грешка при компилация на програмата.

Същите ограничения важат и за не-статичните полета, тъй като техните стойности се съдържат само в инстанциите на класа. В примера това е демонстрирано на ред 11 и на ред 17. И в двата случая статичен метод се опитва да достъпи не-статично поле.

Статичните методи могат да викат други статични методи, както и да осъществяват достъп до статични променливи без проблеми. Това е демонстрирано на редове 18-19.

По-интересната част на демонстрацията е на следващите три реда, където можем да видим, че статичните методи все пак могат да извикват и достъпват не-статични методи и полета. Необходимо условие за това е те да предоставят обект, върху който да бъдат изпълнени тези методи, или респективно - от който могат да бъдат взети тези полета.

Интересно е да отбележим, че по отношение на достъпа статичните методи имат достъп до **private** полетата и методите на класа, защото, разбира се, са дефинирани в същия клас. Така например статичният метод **main** без

проблеми осъществи достъп до **private** полето **test** на ред 21 (чрез обекта **m**), както и до **private** полето **test2** на ред 19.

По отношение на *изписването на имената на не-статичните полета и методи* обаче, статичните методи по-скоро изглеждат като външни за класа методи, тъй като трябва винаги да подадат инстанция на обект.

1.2. Инициализационни блокове и директна инициализация

В предходната глава ние запознахме читателя с конструкторите, които се използват за инициализиране на полетата на обектите. Конструкторите са основен начин за инициализация на инстанциите на класовете в Java, но те се допълват от два други начина, които се използват в някои специфични случаи.

Първият и по-важният от тези начини е директната инициализация. Задаването на директна инициализация на полета е идентично с това при локалните променливи – директно след типа и името на полето в неговата декларация се поставя знак за присвояване, следван от начална стойност за променливата.

Вторият начин е чрез използване на инициализационни блокове, които имат две форми. Директната инициализация и двете форми на инициализационни блокове са демонстрирани на следващата разпечатка.

```
1 class WebDocument {
2     int number;
3     static String url;
4     int error = 404;    // Директна инициализация
5
6     public static String content;
7
8     // Инициализационен блок
9     {
10         number = 5;
11     }
12
13     // Статичен инициализационен блок
14     static {
15         url = "www.test.com";
16     }
17 }
```

Инициализационните блокове, демонстрирани на редове 8-16, са блокове от код, които са разположени директно в класа, а не в някой от неговите методи. Съществуват два вида инициализационни блокове – статични и не-статични. Също като статичните методи и променливи, статичните инициализационни блокове са маркирани от ключовта дума **static**, разположена непосредствено преди началото на блока.

Също като конструкторите, инициализационните блокове могат да съдържат всякакъв програмен код, но на практика кодът в тях се използва за инициализация полетата на класа, и то в специфични и редки случаи. Един клас може да има множество статични и не-статични инициализационни блокове.

Статичните блокове се изпълняват при зареждането на класа от виртуалната машина на Java и преди изпълнението на инициализационните блокове и конструкторите на отделните инстанции. В този смисъл те са първи в реда на инициализацията. Ако даден клас има множество инициализационни блокове, те се изпълняват в реда, в който са написани в изходния код.

Не-статичните инициализационни блокове се изпълняват при създаването на конкретна инстанция, преди изпълнението на нейния конструктор. Ако има няколко инициализационни блока, те се изпълняват в реда, в който са дадени в кода.

Последният начин за инициализация, който ще разгледаме е „директната“ инициализация, при която след типа и името на полето добавяме оператора за присвояване `=` и задаваме стойност. Този начин е демонстриран на ред 4. Читателите вероятно ще забележат, че начинът, по който инициализираме директно полетата, е идентичен на начина, по който инициализираме локални променливи.

Основният оставащ въпрос, свързан с инициализационните блокове и директната инициализация, е това защо е нужно те да съществуват в Java и в какви случаи те са полезни.

Директната инициализация е особено полезна за статичните полета, и особено за полетата-литерали. Тя позволява повечето литерали да бъдат дефинирани на един-единствен ред в класа и логиката по тяхното инициализиране да не се „разпръсква“ и към конструктора на класа.

Добра практика при организацията на програмния код е статичните полета, независимо дали са литерали или не, да се инициализират на място а не в конструктора на класа, който работи най-добре за инициализация на не-статичните полета. Понякога (много рядко) обаче инициализацията на дадено статично поле е по-сложна и не може да бъде направена на един ред. Точно тогава е добра идея да се използва статичен инициализационен блок, който играе ролята на „конструктор“ за статичните полета.

Не-статичните инициализационни блокове са най-редкият вариант на инициализация, поради това, че техните функции дублират тези на конструктора на класа. Те са полезни при инициализиране на анонимни класове, тъй като тези класове не могат да имат конструктор. Също така инициализационните блокове понякога се използват при класове с множество сложни алтернативни (овърлоуднати) конструктори. В този случай в инициализационния блок се поставя онзи инициализационен код, който трябва да се изпълни, независимо кой конструктор е извикан.

1.3. Изброими типове

Много често в програмите се налага да съхраняваме информационни атрибути, които имат фиксиран брой възможни стойности. Така например според паспортите в България, цветовете на очите на гражданите са само 6 вида – черни,

кафяви, сини, сиви, зелени и пъстри. Семейното положение на мъжете пък е само четири вида – женен, неженен, вдовец и разведен.

За да представим подобни атрибути в миналото много често използваме цифри и обикновени цифрови типове. Така например възможните семейни положения вероятно щяха да бъдат представени с цифрите от 0 до 3, т.е. 0-женен, 1-неженен, 2-вдовец и 3-разведен. Използването на цифри по подобен начин беше неудобно в много отношения. Един от най-големите проблеми с този подход бе това, че компилаторът нямаше начин да определи кои цифри са валидни стойности за даден атрибут и кои – не. Нищо не пречеше например програмистът да постави числото 5 в полето за семейно положение и тази грешка щеше да бъде открита чак по време на изпълнението на програмата, вероятно след като е причинила проблеми с обработката на данните.

За да подобрят ситуацията, модерните програмни езици като Java предлагат възможности за дефиниране на *изброими типове*. Тези типове имат точно определени стойности, които са зададени от програмиста. Именно защото всички възможни стойности на изброимия тип са известни и вписани в програмата, Java може да открие кога в променлива от този тип се поставя невалидна стойност още по време на компилация.

Създаването на изброими типове става с ключовата дума **enum**. Долната програма показва създаването и използването на елементарен изброим тип:

```
1 enum EyeColor {BLACK, BROWN, BLUE, GREEN, GRAY, HAZEL}
2
3 public class Main {
4     public static void main(String[] args) {
5         EyeColor c = EyeColor.BLACK;
6         c = EyeColor.BROWN;
7         // c = 1;                                //Грешка при компилация
8         // c = BROWN;                            //Грешка при компилация
9         // boolean q = EyeColor.BLACK == 1;      //Грешка при компилация
10        System.out.println(c);                  // BLACK
11    }
12 }
```

Разпечатка 9.4. Деклариране и използване на изброим тип

В програмата от разпечатка 9.4, изброимият тип **EyeColor** е създаден на ред 1 с помощта на ключовата дума **enum**, следвана от името на типа. След името на типа във фигурни скоби и разделени със запетайки са изброени възможните стойности, които могат да приемат променливите от този тип. Така променлива от тип **EyeColor** ще може да приема точно 6 стойности, а именно **EyeColor.BLACK**, **EyeColor.BROWN**, **EyeColor.BLUE**, **EyeColor.GREEN**, **EyeColor.GRAY** и **EyeColor.HAZEL**.

Както е видно, синтаксисът на **enum** декларацията е много сходен с този на декларацията **class**. Както ще видим по-нататък, това не е случайност.

Ред 6 на разпечатката демонстрира как се декларира променлива от изброим тип. Нещо повече, ред 5 демонстрира, че когато се използват в програмата като литерали, стойностите на изброимите типове винаги се изписват префиксирани с името на типа. Тоест, ние присвояваме на променливата с стойността **EyeColor.BLACK**. Ако бяхме записали тази стойност само като **BLACK**, както е направено на ред 8, това щеше да доведе до грешка при компилация.

За разлика от интерпретацията им в много други езици, стойностите на изброимите типове в Java не могат да се интерпретират директно като числа. В програмите на Java не е позволено число да се превръща директно в стойност на изброим тип и обратното.

Тази важна особеност е демонстрирана на ред 7. Този ред ще доведе до грешка при компилация, защото програмистът се опитва директно да присвои числов литерал (числото едно) на променлива от тип **EyeColor**. По същия начин сравнението между стойността **EyeColor.BLACK** и числото едно, извършено на ред 9, е недопустимо, поради което този ред също ще генерира грешка при компилация.

Вместо да асоциира стойностите на изброимите типове директно с числа, Java третира изброимите типове като специални класове. Точно като при класовете, Java позволява изброимите типове да имат конструктори, методи и

полета. Стойностите на изброимите типове всъщност са инстанции на тези класове.

Това, че стойностите на изброимите типове са обекти, означава, че при нужда към тях могат да се прикачат не просто числа, а всъщност произволна по своята структура информация. Следващият пример демонстрира точно как става това:

```
1  enum EyeColor {BLACK(0, "Черни"), BROWN(1, "Кафеви"), GRAY(2, "Сини"),
2      BLUE(3, "Сини"), GREEN(4, "Зелени"), HAZEL(5, "Пъстри");
3      public int number;
4      private String description;
5
6      EyeColor(int number, String description) {
7          this.number = number;
8          this.description = description;
9      }
10
11     public String getDescription() {
12         return description;
13     }
14 }
15
16 public class Main {
17     public static void main(String[] args) {
18         System.out.println(EyeColor.BLACK.getDescription());
19         System.out.println(EyeColor.HAZEL.number);
20         System.out.println(EyeColor.BLUE.name());
21         System.out.println(EyeColor.valueOf("BLUE").getDescription());
22         // EyeColor e = new EyeColor(); // Грешка при компилация
23         // EyeColor.BLACK = new EyeColor(5, "СИН"); // Грешка при компилация
24     }
25 }
```

Разпечатка 9.5. Използване на изброим тип

Програмата от разпечатка 9.5 разширява изброимия тип, показан на разпечатка 9.4, като добавя към списъка със стойности конструктор (редове 6-9), полета (редове 3 и 4) и метод (редове 11-13). Програмата „прикача“ към всяка стойност на изброимия тип два аргумента – последователен номер на стойността и нейният превод на български език.

Първият и най-лесен за изпускане детайл от този „разширен“ синтаксис на **enum** е това, че списъкът със стойности е отделен от останалите елементи на декларацията с **;** (точка и запетая). Точката и запетаята се намират точно в края

на ред 2 и имат задължителен характер. Оттам нататък синтаксисът е идентичен с този на обикновенните класове.

Тъй като Java третира всяка стойност на изброимия тип като заявка за създаване на обект, прикачените аргументи ще бъдат предадени към конструктора на изброимия тип. В нашия пример Java ще създаде точно 6 инстанции на изброимия тип – по една за всяка негова възможна стойност.

За разлика начина на работа с обикновените класове, програмистът не трябва да създава всяка от стойностите „ръчно“ с **new**. Конструкторът на всяка инстанция (стойност) ще бъде извикан *автоматично* от зареждащия механизъм за класове на виртуалната машина. Нещо повече, ако програмистът се опита да създаде инстанция на **enum** стойност ръчно, както е показано на редове 22 и 23, това ще генерира грешка при компилация.

Читателят трябва да обърне внимание на редове 17-20, които демонстрират как може да се осъществи достъп до полетата и методите на отделните стойности на изброимия тип. Следва да се отбележи, че променливите и методите на изброимия тип могат да имат всички модификатори за достъп, които имат полетата и методите на обикновените класове.

1.4. Анонимни класове, локални класове, вътрешни класове и статични вложени класове

В сложните програми много често възниква нужда определена функционалност да се отдели в собствен клас, за да се структурира по-добре програмата. Създадените по този начин класове често са много тясно свързани с оригиналния *обхващащ* клас и много често се използват само от него.

За такива случаи Java предлага четири начина, по които даден клас може да се „вгради“ в оригиналния. Тези начини именно са анонимните, локалните, вътрешните и статичните вложени класове.

3.1. Статични вложени класове

От четирите типа „вградени“ класове, статичните вложени класове са най-лесни за усвояване от начинаещите програмисти. Също като останалите „вградени“ класове, кодът на статичните вложени класове се помещава вътре в друг клас. Ние ще наричаме такъв клас с не особено стандартизираното име „*обхващащ клас*“.

Статичните вложени класове имат задължителния модификатор **static** пред дефиницията на класа и не могат да се помещават вътре в самите методи на обхващащия клас. За разлика от обикновените класове, статичните вложени класове могат да декларират без ограничения и четирите нива на достъп, които са типични за променливите и методите на класа.

Долният пример демонстрира декларацията на статичен вложен клас:

Student.java	
1	public class Student {
2	public static class Mark {
3	private int no; // Пореден номер
4	private Student student; // Студент
5	private String subject; // Предмет
6	private int digit; // Оценка
7	
8	public Mark(String subject, int digit) {
9	this .subject = subject;
10	this .digit = digit;
11	}
12	
13	public void test() {
14	// marks[1]=null; //Грешка при компилация
15	// fn = 1; //Грешка при компилация
16	String name = student.name;
17	}
18	}
19	
20	private int fn; // Факултетен номер
21	private String name; // Име
22	private Mark[] marks; // Оценки
23	private int markCounter; // Брояч
24	
25	public Student(int fn, String name) {
26	this .fn = fn;
27	this .name = name;
28	this .marks = new Mark[50];
29	this .markCounter = 0;

30	}
31	
32	public void addMark(Mark m) {
33	m.no = markCounter;
34	m.student = this;
35	marks[markCounter++]=m;
36	}
37	}
Main.java	
1	public class Main {
2	public static void main(String[] args) {
3	Student.Mark m = new Student.Mark("Диференциална геометрия", 6);
4	Student s = new Student(1022144, "Иван Атанасов");
5	s.addMark(m);
6	}
7	}

Разпечатка 9.6. Вложен статичен клас

Програмата на разпечатка 9.6 съдържа три класа – **Student**, **Main** и **Mark**. **Main** съдържа входната точка на програмата, а класът **Mark** е вложен статичен клас, поместен в обхващащия клас **Student**. Декларацията на **Mark** започва на ред 2 и завършва на ред 18 в **Student.java**.

Класовете **Mark** и **Student** съдържат опростени данни за студенти и техните оценки. **Mark** съдържа полета за предмет и съответстващата оценка, но също и полета за два други информационни атрибута. Това с полето **Student** на ред 4 на **Student.java**, което съдържа препратка към студента, който е получил оценката, както и полето **no** на ред 3 във същия файл, което съдържа пореден номер на оценката.

В случая разработчикът е решил, че класът **Mark** е достатъчно малък и достатъчно тясно свързан със **Student**, за да бъде реализиран като статичен вложен клас. Конструкторът на **Mark** попълва полетата за предмет и оценка, но полетата **Student** и **no** се попълват едва когато дадена оценка се добави към

оценките на студента с метода **addMark** на **Student** (редове 32-36 на **Student.java**).

Освен като демонстрация на декларирането на статичен вложен клас, примерът е замислен и като препратка към най-важните свойства на статичните вложени класове. Тези свойства са няколко:

На първо място, *инстанциите* на вложените статични класове могат да се използват напълно независимо от инстанциите на обхващащия клас. От гледна точка на ползвателя му, вложеният клас е просто друг клас с по-странно име.

Читателите могат да разгледат кода в метода **main** и да се убедят, че инстанцията на **Student.Mark** се създава преди инстанцията на класа **Student**. По наблюдателните читатели ще забележат, че когато посочваме статичния вложен клас от обхващащия клас не се налага да префиксираме името на статичния вложен клас с каквото и да било. Това може да се види на редове 22 и 28 от **Student.java**, където използваме името на класа **Mark** като тип на елементи от масив. Във всички други случаи ние трябва да префиксираме името на статичния вложен клас с името на обхващащия клас. Такова префиксиране може да бъде видяно на ред 3 на **Main.java**.

Липсата на връзка между инстанциите на вложения и обхващащия клас води и до **второто важно свойство** на статичните вложени класове. **Инстанцията на статичен вложен клас не може да ползва директно полета и методи на обхващащия клас.** Директното използване на методи и полета предполага наличие на конкретна инстанция на обхващащия клас, която е асоциирана с инстанцията на вътрешния клас. Тъй като такава липсва, подобно директно използване е невъзможно.

В нашия пример, евентуален код в класа **Mark** може да осъществи достъп до до полетата и методите на класа **Student**, само ако програмистът му предостави реална инстанция на този клас. При това модификаторите за достъп на полетата на **Student** трябва да позволяват достъпа от класове в същия пакет.

Двете особености могат да се обобщят по следния начин: кодът на статичните вложени класове е вграден в други класове, за да се подобри инкапсулацията и четливостта на програмата. Но отделно от това, статичните вложени класове се използват като съвсем независими класове. Техните инстанции не са обвързани с инстанциите на класовете, в които са поместени.

На този етап читателите вероятно се питат кога един обслужващ клас трябва да бъде вложен и кога е по-добре просто да бъде деклариран като обикновен клас с видимост по подразбиране. На този въпрос трудно може да бъде даден точен отговор и до голяма степен изборът зависи от предпочитанията на програмиста.

Авторът се придържа към практиката да реализира като статични вложени класове само такива, класове, които едновременно са малки (няколко реда) и тясно свързани с техния основен ползвател. Такива могат да бъдат класове, представляващи събития или пък поделементи на сложен визуален GUI елемент, `enum` изброявания и пр.

3.2. Вътрешни класове

Също като статичните вложени класове, кодът на вътрешните класове се помещава вътре в друг обхващащ клас. За разлика от статичните вложени класове, **вътрешните класове обаче съществуват само във връзка с реална инстанция на обхващащия клас**. Поради това те предлагат много по-тясна интеграция и имат някои необичайни ограничения.

Вътрешните класове имат пълен достъп до променливите и методите на инстанцията на обхващащия клас, в рамките на която са създадени. Това е така дори когато тези променливи и методи са маркирани като `private`. С други думи вътрешните класове не просто се помещават в обхващащия ги клас за

подобряване на четливостта на кода, но техните инстанции всъщност стават неделима част от инстанцията на обхващащия клас.

Понеже всяка инстанция на вътрешен клас е свързана с конкретна инстанция на обхващащия клас, създателите на Java са избрали те да нямат статични методи и променливи.

2. Разширено скриване

Читателите едва ли са забелязали, но в примерите досга ние демонстрирахме модификаторите за достъп в два различни контекста.

В първата ситуация ние достъпвахме полета и извиквахме методи директно от методите на класа-наследник, като по този начин указвахме, че се интересуваме от методите и полетата на *текущия* обект, били те наследени или декларирани в класа на текущия обект. **Във втората ситуация** ние имахме променлива, която сочеше към друг обект и ние използвахме променливата, за да достъпим полетата и методите на този обект. Следващата разпечатка експлицитно подчертава разликите между двете ситуации:

```
1 package uk.co.lalev.javabook;
2
3 class Parent {
4     public int t;
5     public void s() {};
6 }
7
8 public class Main extends Parent {
9     public void test() {
10         t=5; //Директен достъп до полето t на текущия обект, наследено от
11             //Parent
12         s(); //Директно извикване на метод на текущия обект
13     }
14
15     public static void main(String[] args) {
16         Parent parent = new Parent();
17         parent.t = 6; //Достъп до поле на друг обект, сочен от
18                     //променливата Parent.
19         parent.s();  //Извикване на метод от друг обект, сочен от
20                     //променливата Parent.
21     }
22 }
```

Разпечатка 8.18. Директен достъп срещу достъп чрез използване на променлива

Директният достъп е възможен само тогава, когато нашият метод достъпва полета и методи от същия обект, независимо дали тези полета и методи са дефинирани в класа на обекта или класа-родител на обекта. На разпечатка 8.18 класът **Main** е наследник на класа **Parent**, което позволява на методите в **Main** да достъпват методите и полетата на **Parent** директно.

Директен достъп до поле на **Parent** е реализиран в метода **test** на **Main** (ред 10), като на следващия непразен ред е демонстрирано и директно извикване на метод. На ред 17 и 19 от друга страна имаме другата ситуация, когато достъпваме полета и методи чрез използване на променлива.

Разграничението до момента не беше важно, тъй като независимо от двата контекста, досега разгледаните модификатори за достъп действат еднакво. Това е демонстрирано още веднъж в разпечатка 8.19.

```
1 package uk.co.lalev.javabook;
2
3 class Test {
4     private int priv;
5     public int pub;
6     int def;
7 }
8
9 public class Main extends Test {
10
11     public void testTheTest() {
12         priv=5; //Директен достъп. private поле. Грешка!
13         pub=6; //Директен достъп. public поле. OK!
14         def=7; //Директен достъп. default поле. OK!
15     }
16
17     public static void main(String[] args) {
18         Test test = new Test();
19         test.priv=5; //Чрез променлива. private поле. Грешка!
20         test.pub=6; //Чрез променлива. public поле. OK!
21         test.def=7; //Чрез променлива. default поле. OK!
22     }
23 }
```

Разпечатка 8.19 Ефекти от досега разгледаните модификатори при директен достъп срещу достъп чрез използване на променлива

Не така стоят нещата с последния модификатор, а именно – **protected**. Този модификатор има доста особености, което именно е причината да го разгледаме последен.

Когато става въпрос за *директен достъп*, правилата за методите и полетата, маркирани с **protected** са доста праволинейни. Полета или методи, които са маркирани **protected**, могат да бъдат достъпвани директно само от класовете в същия клас, както и от класовете-наследници на текущия клас без значение в кой пакет се намират те.

Долната разпечатка демонстрира директен достъп до **protected** методи и полета:

```
1 package uk.co.lalev.javabook;
2
3 class Parent {
4     protected int v;
5     protected void w() { }
6 }
7
8 public class Main extends Parent {
9     public void test() {
10         v=5; //OK
11         w(); //OK
12     }
13
14     public static void main(String[] args) { }
15 }
```

Разпечатка 8.20 Директен достъп до маркирани с **protected** методи и полета

На разпечатка 8.20 класът **Main** е наследник на **Parent**, което му позволява да достъпва полетата и методите директно. Модификаторът **protected** позволява директния достъп, поради което декларациите на редове 10 и 11 се компилират успешно.

Правилата за достъп през променлива са малко по-особени. Също като при директния достъп до **protected** полета и методи, когато се използва променлива за достъп до класа, променливите и методите, на класа, маркирани с **protected**, могат да се достъпват от същия клас и от класовете в същия пакет. Това е демонстрирано на следващата разпечатка:

```

1 package uk.co.lalev.javabook;
2
3 class Test {
4     protected int v;
5     protected void w() {};
6 }
7
8 public class Main {
9
10     public static void main(String[] args) {
11         Test test = new Test();
12         test.v = 5; //OK. Намираме се в същия пакет като този на Test.
13         test.w(); //OK. Намираме се в същия пакет като този на Test.
14     }
15 }

```

Разпечатка 8.21 Достъп до макирани с `protected` методи и полета чрез променлива

В допълнение обаче, и за разлика от ситуацията с директния достъп, ако имаме два класа в отношение родител – наследник, без значение в кой пакет се намират те, съществува възможност да извикваме `protected` методите и полетата на класа родител от методите на класа-наследник през променлива. За целта променливата трябва да е от типа на *класа-наследник* или клас, който е негов наследник надолу по обектната йерархия.

Remote.java	
1	<code>package uk.co.lalev.javabook.test;</code>
2	
3	<code>public class Remote {</code>
4	<code> protected int v;</code>
5	<code> protected void w() { }</code>
6	<code>}</code>
Main.java	

1	package uk.co.lalev.javabook;
2	
3	import uk.co.lalev.javabook.test.Next;
4	import uk.co.lalev.javabook.test.Remote;
5	
6	public class Main extends Remote {
7	public void test() {
8	v=5; //OK.Директен достъп от наследник към protected поле на родител
9	w(); //OK.Директен достъп от наследник към protected поле на родител
10	}
11	
12	public static void main(String[] args) {
13	Remote p = new Remote();
14	p.v=5; //Грешка!
15	
16	Main main = new Main();
17	main.v=5; //OK!
18	
19	Remote main1 = new Main(); //!!!! Полиморфизъм
20	main1.v=5; //Грешка
21	
22	Next next = new Next();
23	next.v=6; //OK!
24	}
25	}
Next.java	
1	package uk.co.lalev.javabook.test;
2	
3	import uk.co.lalev.javabook.Main;
4	
5	public class Next extends Main { }

Разпечатка 8.21 Достъп до макрирани с **protected** методи и полета чрез променлива

Програмата в рапечатка 8.21 се състои от три класа, които изграждат обектна йерархия. **Remote** е клас-родител на **Main**, а **Main** е родител на **Next**. **Remote** и **Next** са в различен пакет от **Main**, така че правилата за достъп до **protected** полета и методи от класове в същия пакет не важат.

На редове 7 и 8 в **Main.java** е демонстрирано, че тъй като **Main** е наследник на **Remote**, неговите методи могат да достъпват **protected** методи и полета на **Remote** директно.

Много по интересен случай е демонстриран на ред 14. На този ред използваме променлива от тип **Person**, която сочи към обект от същия тип. Чрез променливата правим опит да достъпим **protected** полетата и методите на **Person**, но това е неуспешно, тъй като не е позволено в **Java**, дори при

положението в примера, при което `Main` е наследник на `Person`. Правилото, формулирано по-горе, казва друго – възможно е да достъпим `protected` методите и полетата на `Person` от метод във `Main` чрез променлива, но ако става въпрос за променлива от тип `Main` или тип-наследник на `Main`.

Именно това е демонстрирано на редове 16 и 17 в `Main.java`. На ред 16 създаваме променлива от тип `Main`, която сочи към новосъздаден обект от същия тип. На ред 17 използваме тази променлива, за да достъпим `protected` поле, наследено от `Remote`.

Читателите може би забелязаха, че в рамките на последната страница сменихме формулировката и започнахме да уточняваме вида на обектите, към които сочат променливите от нашите променливи от референтен тип. Това е така, защото постепенно се приближаваме към последния механизъм на обектно-ориентираното програмиране, за който старателно избягвахме да говорим до момента. Благодарение на полиморфизма, който ще разгледаме след малко, можем да имаме и ситуации, при които променлива от типа-родител съдържа обект от типа-наследник. Тъй като типът наследник има всички полета и методи на родителя (макар и евентуално предефинирани), той може да замести типа-родител навсякъде където се изисква това.

Ситуацията на ред 19 е такава. Създаваме обект от тип `Main`, но го присвояваме на променлива от тип `Remote`. Правим това, за да определим дали променливата или типа на самия обект определят това дали ще имаме достъп до `protected` методите и полетата, наследени от `Remote`.

Както се вижда от ред 20, променливата определя правата за достъп, а не конкретния указван обект. Ето защо имаме ситуация, аналогична на тази на ред 14 – грешка при компилация.

Последният експеримент касае използването на променлива от тип клас-наследник на `Main`. На редове 22 и 23 това е класът `Next`. Както е видно от примера, достъпът до `protected` методите и полета, наследени от `Parent` и след това от `Main`, е позволен по този начин.

За програмистите, които се опитват да „наредят“ в главата си правилата за достъп до `protected` полетата е полезно да направим обобщение. Полетата и методите, маркирани `protected`, основно са предназначени за директно използване от класовете-наследници. Но има определени операции, които налагат изключения. Такива операции изискват метод в класа да работи върху инстанции на същия клас или инстанции на класовете-наследници. Такава операция например би била копирането на обект в друг от същия тип. Това предполага и изключението, което бе демонстрирано току-що.

Долната таблица обобщава ефектите от модификаторите за видимост при *директен достъп*:

	достъп от същия клас	достъп от клас в същия пакет /наследник или не/	достъп от клас в друг пакет, който е наследник	достъп от клас в друг пакет, който не е наследник
<code>public</code>	да	да	да	да
<code>protected</code>	да	да	да	не
(по подразбиране)	да	да	не	не
<code>private</code>	да	не	не	не

Табл. 8.2. Модификатори за достъп до методи и полета на клас в Java при директен достъп

Следващата таблица пък обобщава ефектите от модификаторите за видимост при достъп чрез променлива:

	достъп от същия клас	достъп от клас в същия пакет /наследник или не/	достъп от клас в друг пакет, който е наследник; променливата е от типа на класа наследник или под него в обектната йерархия	достъп от клас в друг пакет, който е наследник; променливата е от типа на наследявания клас	достъп от клас в друг пакет, който не е наследник
<code>public</code>	да	да	да	да	да
<code>protected</code>	да	да	да	не	не

(по подразбиране)	да	да	не	не	не
private	да	не	не	не	не

Табл. 8.3. Модификатори за достъп до методи и полета на клас в Java при достъп чрез променлива

Завършваме настоящата точка с наблюдението, че модификаторите за достъп могат да се прилагат и върху цели *класове*. Ние видяхме множество примери за това в разпечатките досега. Време е да споменем обаче, че имаме два вида класове – такива, които са поместени директно в пакет, и такива, които са *вложени в други класове или методи*. Досега не сме се срещали с втория вид класове, но е редно да кажем, че те могат да получават всички споменати модификатори за достъп. Ние ще разгледаме ефектите от тези модификатори когато разгледаме вложените класове по-нататък.

Всички класове, които наблюдавахме до момента, са директно поместени в пакети. Ние наричаме такива класове „**класове от най-горно ниво**“. Това понятие няма нищо общо с обектната йерархия на наследяването, а касае къде е поместен този клас. Класовете от най-горно ниво могат да получават само два модификатора – `public` и „по подразбиране“.

Класовете от най-горно ниво, маркирани като `public`, могат да се ползват от класове извън пакета. Класовете, маркирани като `private`, могат да се използват само от същия пакет. Долният пример демонстрира тези разлики:

Accessible.java	
1	<code>package uk.co.lalev.javabook.test;</code>
2	
3	<code>class Internal { }</code>
4	
5	<code>public class Accessible { }</code>
6	
7	<code>public class Extra { } //Грешка при компилация</code>
Main.java	

```

1 package uk.co.lalev.javabook;
2
3 import uk.co.lalev.javabook.test.Accessible;
4 import uk.co.lalev.javabook.test.Internal; //Грешка!
5
6 public class Main extends Internal { //!!!
7     public static void main(String[] args) {
8         Accessible a = new Accessible();
9     }
10 }

```

Разпечатка 8.22 Модификатори за достъп, приложени към клас от най-горно ниво

Програмата на разпечатка 8.22 е съставена от два файла, които поместват няколко класа с различни модификатори за достъп. Програмата генерира целенасочено няколко грешки, за да демонстрира казаното по-горе. Първата грешка се намира на ред 7 в `Accessible.java`. Дори без да дискутираме ефектите на модификаторите към класовете, много пъти вече споменахме, че класове от най-горно ниво, маркирани като `public`, трябва да се намират в отделен файл, който има същото име като класа и разширение `.java`. В случая декларацията на клас `Extra` води до грешка именно по тази причина.

Към класовете с разрешения за достъп по подразбиране няма такива ограничения. Те могат да бъдат дефинирани сами във файл или пък (както е в случая) да бъдат дефинирани във файловете на други класове.

Другата грешка е на ред 4 в `Main.java`. Грешката е породена от опит за импортиране на клас, който има видимост по подразбиране и се намира във външен пакет.

ние създавахме клас, в който да поставим метода *main*. Все пак обаче горните обяснения могат да прозвучат на читателя малко абстрактно, тъй като досега ние не сме писали истински класове с истински функции, като например изчисляване на ДДС, дебитиране на сметка и пр.

Ето защо имаме нужда от достъпна и убедителна демонстрация за ползите от инкапсулирането на методи заедно с данните в един клас. Започваме тази

демонстрация, връщайки читателите към разпечатка 8.2, която показва създаването на променливи от тип *Person*. Този път ние ще зададем нов въпрос. Какво ще стане ако програмистът, който попълва полетата на обекта с начални стойности, сгреша като пропусне да въведе стойност за някое от полетата на обекта? Така например програмистът може да забрави да въведе стойност за полето *age* на обекта *secondPerson* или пък полето *name* на *firstPerson*.

Отговорът е, че това със сигурност ще повреди програмата и то вероятно по начин, който няма да бъде открит директно при компилация. В реални ситуации тези грешки са често срещани и произтичат не просто от разсеяност, а понякога от това, че създателят на класа е един програмист, а ползвателят – друг. В такъв случай различни фактори като например неправилно построената, недостъпната или непълната документация или пък липсата на опит на ползвателите за работа с дадената библиотека допринасят за възникването на грешките.

Обектно ориентираното програмиране адресира този проблем с помощта на специален метод, който се *включва* във *всеки* един клас. Това е т.нар. „*конструктор*“. Ролята на конструктора е да попълни всички задължителни полета още при създаването им, проверявайки в процеса всички начални данни за валидност.

Долната разпечатка демонстрира дефинирането на конструктор към класа *Person* от разпечатка 8.2.

```
1  class Person {
2      String name;
3      String surname;
4      String family;
5      int age;
6
7      Person(String newName, String newSurname, String newFamily, int newAge){
8          name = newName;
9          surname = newSurname;
10         family = newFamily;
11         age = newAge;
12     }
13 }
14
```

```

15 public class Main {
16     public static void main(String[] args) {
17         Person person = new Person("Йосиф", "Андреев", "Петров", 47);
18         System.out.println(person.name);           // Йосиф
19         System.out.println(person.surname);         // Андреев
20         System.out.println(person.family);          // Петров
21         System.out.println(person.age);             // 47
22     }
23 }

```

Разпечатка 8.3 Използване на класове

Конструкторът заема редове 7-12 на разпечатката. **В Java конструкторите не могат да връщат стойност и техните имена трябва да съвпадат с името на класа, който ще бъде създаван.** Ние можем да разпознаем дефиницията на даден метод като конструктор именно по тези два признака. Така например в дефиницията на ред 7 нямаме връщания тип. Обикновен метод би посочил този тип като *int*, *String* или *void*. Вместо това дефиницията започва направо с името на конструктора, което съвпада с името на обекта. Съвпадението на имената е именно и вторият признак, по който разпознаваме конструктора.

Конструкторът обаче може да взема аргументи като всеки друг метод. Нашият конструктор взема четири аргумента – *newName*, *newSurname*, *newFamily* и *newAge*. Той присвоява тези параметри на съответните полета и по този начин инициализира обекта по валиден начин. Интересно обаче е как конструкторът получава реалните стойности за своите аргументи. Това става когато обектът се създаде с конструкцията *new*. Както е видно от ред 17, в резултат от извикването на конструктора, параметърът *newName* ще получи стойност „Йосиф“, който ще бъде поставен (благодарение на кода на ред 8) в полето *name*. По същия начин стойностите „Андреев“, „Павлов“ и 47 ще бъдат поставени съответно в полетата *surname*, *family* и *age*.

Редовете от 8 до 11 демонстрират важно свойство на обектите. Когато методите на даден обект трябва да осъществят достъп до полетата на обекта, това става като при „обикновените“ променливи. Достатъчно е да се напише само името на полето. Така например ред 11, част от конструктора, инструктира

виртуалната машина на Java да вземе стойността на параметъра *newAge* и да я постави в полето *age*.

Веднага се връщаме към разпечатка 8.2, за да демонстрираме че когато методът не се числи към дадения обект, начинът за достъп е различен. Така например на ред 13 на разпечатка 8.2, методът *main* на класа *Main* осъществява достъп до полето *name* на променливата *firstPerson*. Тъй като *firstPerson* е от различен клас, името на полето *name* трябва да се префиксира с името на променливата, както е направено на ред 13.

Наличието на методи създава ясно разграничение между два типа променливи. От една страна съществуват променливите, които са дефинирани вътре в даден метод и които използвахме навсякъде до момента. От друга страна полетата не се намират в никой метод, а са част от самия клас.

За да различаваме променливите, дефинирани в методите от полетата на класа, ние често ги наричаме „**локални променливи**“.

Като цяло, между работата с двата вида променливи има малко разлики. Все пак трябва да се помни следното:

1. Локалните променливи са достъпни само в метода, в който са дефинирани. Полетата са достъпни за методите на целия клас, а понякога и за методи, които са разположени в други класове.
2. Локалните променливи трябва да получат начална стойност, зададена от програмиста.

Маркирането на полетата като **private** и използването на гетери и сетери дава гаранции, че индивидуалното поле ще се инициализира винаги с проверени за коректност данни. Какво ще се случи обаче, ако програмистът *въобще пропусне* да постави стойност в някое поле със задължителен характер? Такова изпускане всъщност е направено на разпечатка 8.15, когато програмистът инициализира *person* с възраст, но не и име.

Методите-конструктори са начинът за противодействие на този проблем при обектно-ориентираното програмиране. Те имат за задача инициализирането на

полетата на новосъздадения обект с валидни стойности, така че към момента на приключване на метода, обектът да е валиден и готов за употреба. За да се предотврати възможността програмистът да създаде обекта и после да забрави да извика конструктора, в Java и всички модерни програмни езици конструкторите се извикват едновременно със създаването на обекта.

Конструкторите са особени методи, тъй като трябва да работят добре с ключовата дума `new`. Това изискване налага две важни особености на конструкторите. **Конструкторите не могат да връщат стойност и техните имена трябва да съвпадат с името на класа, който ще бъде създаван.**

Долният пример демонстрира използването на конструктор:

```
1 package uk.co.lalev.javabook;
2
3 class Person {
4     private String name;
5     private int age;
6
7     public Person(String name, int age) {
8         this.name = name;
9         this.age = age;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public int getAge() {
17        return age;
18    }
19 }
20
21 public class Main {
22     public static void main(String[] args) {
23         Person person = new Person("Иван Петров Ангелов", 24);
24         System.out.println(person.getAge()+" "+person.getName());
25     }
26 }
```

Разпечатка 8.27 Дефиниране на метод-конструктор

В разпечатка 8.27 конструкторът се намира на редове 7-10. Докато методите могат да изберат между това да връщат и да не връщат стойност (за последното те сигнализират с ключовата дума `void`), конструкторите никога не връщат

стойност, поради което не се нуждаят от позиция за нея в дефиницията си. Ето защо декларираният от нас конструктор има само модификатор за видимост. Примерът от разпечатката демонстрира и една стандартна практика - параметрите на конструктора имат същите имена като тези на полетата в класа. Вътре в конструктора тези имена влизат в конфликт, поради което имената на полетата трябва да се префиксират с ключовата дума `this`.

На ред 23 в разпечатката читателите също трябва да забележат как конструкторът се извиква заедно със създаването на обекта. За целта след ключовата дума `new` се изписва името на класа/конструктора (те са винаги еднакви в Java) като в скобите се поставя списъкът с параметри.

Важен за отбелязване факт е и това, че наличието на конструктори позволява създаването на *неизменчиви*¹ обекти. Полетата на такива обекти се инициализират при създаване с конструктора. Тъй като класовете на тези обекти имат гетери, но не и сетери и тъй като методите на обекта не променят данните в полетата, съдържанието на полетата не може да се променя по времето на живота на обекта. Често алгоритмите за работата с изменчиви обекти могат да се преформулират в алгоритми за работа с неизменчиви обекти. Резултатът от този процес е това, че обработката на данните се свежда до трансформиране на един неизменчив обект в друг такъв неизменчив обект, създаден на база данните от първия. Този начин на работа носи предимства, когато например извършваме високопроизводителна многопоточна обработка на данните, ангажираща всички ядра на процесора на компютъра.

¹ От англ. „immutable”.

Читателите вероятно вече са се досетили, че класовете и обектите са начинът в Java, по който осъществяваме инкапсулацията. Класовете могат да бъдат давани за тип на променливи и масиви, което ни позволява да мислим за данните си не като разпиляни в различни структури елементарни типове, а като *обекти*, които съответстват на „обектите“ от реалния свят. Нашите обекти имат полета, които обясняват какви са важните за нашата програма информационни атрибути на обектите от реалния свят, както и методи, които обясняват какви действия могат да бъдат извършвани върху тези атрибути.

3. Модификатори за достъп в Java

В Java съществуват четири модификатора за достъп, които могат да бъдат прилагани към различни езикови конструкции. Тези модификатори реализират още един механизъм, свързан с обектно-ориентираното програмиране, а именно – скриването.

Модификаторът `private`, поставен пред дефиницията на метод или поле в даден клас, означава, че този метод или поле няма да може да се използва *извън* дадения клас. В противовес, модификаторът **`public`** позволява достъп от всеки клас, независимо в кой пакет е разположен той. Долният пример демонстрира използването на `private` и `public` модификатори.


```

1 package uk.co.lalev.javabook;
2
3 class Person {
4     public String name;
5     public String middleName;
6     public String surname;
7     private int age;
8 }
9
10 public class Main {
11     public static void main(String[] args) {
12         Person person = new Person();
13         person.name="Нина";
14         person.middleName="Лазарова";
15         person.surname="Лалева";
16         person.age = 50; //Грешка при компиляция
17     }
18 }

```

Разпечатка 8.14 Използване на модификатори `public` и `private`

В примера полетата `name`, `middleName` и `surname` са отбелязани с модификатор `public`, което означава, че те могат да бъдат достъпвани от други класове. За да сме по-точни (тъй като в Java програмният код може да се намира само в методи, а методите могат да се намират само в класове) можем да кажем, че тъй като полетата са маркирани `public`, те могат да се използват от методи, принадлежащи на произволни други класове, независимо в кой пакет се намират тези класове и какво е отношението им към `Person`. Това е причината, поради което методът `main` от класа `Main` може да осъществи достъп до тези полета без проблеми на редове 13-15.

Модификаторът `private` от друга страна забранява използването на полето `age` от методи, разположени в други класове, без значение къде са разположени тези класове и какво е отношението им към класа `Person`. Тъй като методът `main` е разположен извън класа `Person`, той не може да достъпи полето `age`, което води до грешка по време на компиляция.

Читателите, които тепърва се сблъскват с модификаторите за достъп и скриването, най-вероятно лесно ще си представят защо можем да декларираме даден *метод* `private`. Нуждата от правилна организация и избягване повторното писане на код водят до това, че в класовете много често имаме различни помощни

методи, които съществуват само за да подпомогнат работата на основните методи на класа. Когато програмистът прецени, че няма причини (и/или полза) такива методи да се извикват директно от външни класове, той обикновено маркира тези методи `private`. Така, както споменахме и в предната точка, скриването всъщност улеснява ползвателите на класа, тъй като то ясно маркира кои методи са вътрешни за класа (`private`) и кои методи всъщност са начините, по които външни класове следва да използват класа (`public`).

Малко по-объркващ за начинаещите е демонстрираният случай на полета, маркирани `private`. Защо бихме искали да ограничим достъпа на външни класове до дадено поле и как тогава ние можем да поставим стойност в него? Най-често причината за подобно решение се корени в нуждата на програмиста да контролира смислеността на входните данни.

Връщайки се към пример 8.14 ние виждаме, че полето `age` е от тип `int`. Нищо не пречи в него да поставим например `-60`, въпреки че отрицателните стойности нямат смисъл в конкретния контекст. По същия начин нищо не пречи да поставим празен текст за полето `name`, въпреки че няма хора без собствени имена. Въпреки че подобни грешки изглеждат наивни, когато кодът стане сложен и се сподели между множество програмисти, става много лесно един програмист да направи грешка и да подава безсмислени данни на клас, разработен например от друг програмист. Ситуации с грешни входни данни, към които се прилага иначе коректно описана логика на реализираната задача, се срещат толкова често, че си имат неофициално име - „garbage in – garbage out” („влиза боклук – излиза боклук“). Оказва се, че скриването предлага идеално решение за избягването им. Конкретният начин, по който се прави това, е толкова често срещан в обектно-ориентираните програми, че читателят трябва задължително да го проумее и запомни. Подобни „добри практики“ се наричат **шаблони на дизайна** и са част по-общата и теоретична област на софтуерното инженерство:

```

1 package uk.co.lalev.javabook;
2
3 class Person {
4     private String name;
5     private int age;
6
7     public String getName() { return name; }
8
9     public void setName(String newName) {
10         if (newName.isEmpty()) //Проверява за празен симв. низ
11             throw new RuntimeException(); //Сигнализира за грешка
12         name = newName;
13     }
14
15     public int getAge() { return age; }
16
17     public void setAge(int newAge) {
18         if (newAge<0) throw new RuntimeException(); //Сигнализира за грешка
19         age = newAge;
20     }
21 }
22
23 public class Main {
24     public static void main(String[] args) {
25         Person person = new Person();
26         person.setAge(20);
27         System.out.println(person.getAge());
28         person.setAge(-20); //Грешка!
29         person.setName(""); //Грешка!
30     }
31 }

```

Разпечатка 8.15 Използване на гетери и сетери

Кодът на разпечатка 8.15 представлява модификация на кода от пример 8.14. От съображения за краткост на примера този път класът `Person` има само две полета – `name` и `age`. Полето `name` съдържа име, а `age` съдържа възраст на лицето. И двете полета са маркирани `private`, което означава че външните класове не могат да боравят *директно* с тях. За да позволи на външни класове все пак да задават и изчитат подават стойност, програмистът предоставя по два метода на поле.

За полето `name`, тези методи се казват `setName` и `getName`. И двата метода са публични, което означава, че за разлика от самото поле, те могат да се извикват от външни класове! Предназначението на `setName` е да задава нова стойност на полето `name`. Методът `getName` пък връща стойността на полето `name`.

Изглежда, че благодарение на двата метода, макар и индиректно, външните класове могат да променят и използват стойността на полето `name`. Това, което се печели чрез такъв индиректен достъп до стойността на полето `name`, е възможността в метода `setName` да се поставят проверки за смислеността на новата стойност за полето `name`.

Методът `setName` използва две конструкции, които не сме споменавали до момента в изложението. На ред 10 за пръв път извикваме метод на класа `String`. `isEmpty` проверява дали символният низ е празен. Ако е така, на ред 11 генерираме първото си *изключение*. Изключенията са начинът, по който сигнализираме за неочаквани грешки по време на изпълнението на програмата. Генерирането на изключения така, както е показано в примера, е далеч от идеалното, но все-пак постига основната ни цел. В случая, за да не позволим в програмата да се промъкнат грешни входни данни, при достигане на ред 11 програмата просто ще спре. Когато разгледаме в подробности изключенията (които също са класове) ние ще можем да формулираме доста по-изтънчени стратегии за реагиране и възстановяване от неочаквани грешки.

Точно по същия начин работят методите `getAge` и `setAge`. `getAge` позволява изчитане на стойността, поставена в `private` полето `age`, но отново същинското подобрение е възможността да проверим смислеността на новата стойност за `age` само веднъж, още когато тази нова стойност се поставя в полето чрез метода `setAge`. При отрицателна стойност на параметъра `newAge`, методът `setAge` ще генерира изключение, което ще спре изпълнението на програмата.

Програмистите на Java наричат методите като `setAge` и `getAge` съответно сетери (“setters”) и гетери (“getters”). Читателите следва да забележат, че в много ситуации е полезно да генерират сетери и гетери, дори да нямат нужда от конкретна програмна логика за проверка на смисленост. Това позволява полесното вмъкване на такава логика в класовете-наследници на текущия клас, които ще бъдат евентуално създадени на някакъв по-късен етап.

В началото на точката казахме, че модификаторите за достъп са четири. Казвайки това, ние съзнателно пропуснахме важен детайл. Всъщност модификаторите са само три, от които досега разгледахме два – `public` и `private`. Режимите за достъп обаче са четири. Третият „модификатор“, който ще разгледаме, всъщност е ситуацията, в която *липсва* модификатор за достъп. Липсата на модификатор за достъп определя режим за достъп, който е различен от режимите, зададени с един от трите модификатора. Често този режим се обозначава с думите „по подразбиране“ или “default”.

Ако пред дадени полета или методи на класа липсва модификатор за достъп, до тях може да се осъществява достъп само от същия клас, както и от класове, които се намират в същия пакет. Това е демонстрирано в долния пример:

Test.java	
1	<code>package uk.co.lalev.javabook;</code>
2	
3	<code>public class Test {</code>
4	<code> public int pub;</code>
5	<code> private int priv;</code>
6	<code> int def; //Режим на достъп по подразбиране</code>
7	<code>}</code>
Main.java	
1	<code>package uk.co.lalev.javabook;</code>
2	
3	<code>public class Main {</code>
4	<code> public static void main(String[] args) {</code>
5	<code> Test t = new Test();</code>
6	<code> t.pub = 5; //OK. Достъп до public поле</code>
7	<code> t.priv = 6; //Грешка! Опит за достъп до private поле</code>
8	<code> t.def = 7; //OK. Поле с режим на достъп по подразбиране, достъп</code>
9	<code> //от клас в същия пакет.</code>
10	<code> }</code>
11	<code>}</code>
Remote.java	
1	<code>package uk.co.lalev.javabook.other;</code>
2	
3	<code>import uk.co.lalev.javabook.Test;</code>
4	
5	<code>public class Remote {</code>
6	<code> public void test() {</code>
7	<code> Test t = new Test();</code>
8	<code> t.pub = 5; //OK. Достъп до public поле.</code>
9	<code> t.priv = 6; //Грешка! Опит за достъп до private поле</code>
10	<code> t.def = 7; //Грешка! Поле с режим на достъп по подразбиране. Достъп</code>
11	
12	

13	} }	// от клас извън пакета.
----	--------	--------------------------

Разпечатка 8.16. Демонстрация на режим на достъп по подразбиране

Програмата на разпечатка 8.16 се състои от три класа, като класът `Remote` е дефиниран в различен пакет от останалите два. Класът `test` има три полета с модификатори съответно `public`, `private` и „по-подразбиране“ (тоест липсващ модификатор за достъп).

На редове 6 и 7 в `Main.java` ние опитваме поред достъп до `public` и `private` полето на класа `Test`. Тъй като `public` на практика разрешава достъп безусловно и отвсякъде, ред 6 се компилира успешно. Ред 7 обаче генерира грешка при компилация, тъй като `private` полетата са достъпни само за методите на собствения им клас. Ред 8 демонстрира свойствата на режима на достъп по подразбиране. Тъй като `Main` и `Test` са в един пакет, `Main` може да достъпи без проблем полето и редът се компилира. Не такъв обаче е случаят с ред 10 в `Remote.java`. Там се извършва напълно аналогичен опит за достъп, но този път `Remote` е в друг пакет спрямо `Test`. Резултатът е грешка при компилация на програмата.

Читателите едва ли са забелязали, но в примерите досга ние демонстрирахме модификаторите за достъп в два различни контекста.

В първата ситуация ние достъпвахме полета и извиквахме методи директно. Във втората ситуация ние имяхме променлива, която сочеше към обект. Следващата разпечатка експлицитно подчертава разликите между двете ситуации:

```

1 package uk.co.lalev.javabook;
2
3 class Parent {
4     public int t;
5     public void s() {};
6 }
7
8 public class Main extends Parent {
9     public void test() {
10         t=5; //Директен достъп
11         s(); //Директно извикване на метод
12     }
13
14     public static void main(String[] args) {
15         Parent parent = new Parent();
16         parent.t = 6; //Достъп до поле чрез променлива
17         parent.s(); //Извикване на метод чрез променлива
18     }
19 }

```

Разпечатка 8.17. Директен достъп срещу достъп чрез използване на променлива

Директният достъп е възможен само тогава, когато нашият метод достъпва полета и методи от същия клас или тогава, когато нашият метод се числи към клас-наследник на достъпвания клас. На разпечатка 8.17 класът `Main` е наследник на класа `Parent`, което позволява на неговите методи да достъпват методите и полетата на `Parent` директно.

Директен достъп до поле на `Parent` е реализиран в метода `test` на `Main` (ред 10), като на следващия ред е демонстрирано и директно извикване на метод. На ред 15 и 16 от друга страна имаме другата ситуация, когато достъпваме полета и методи чрез използване на променлива.

Разграничението до момента не беше важно, тъй като независимо от двата контекста, досега разгледаните модификатори за достъп действат еднакво. Това е демонстрирано още веднъж в разпечатка 8.18.

```

1 package uk.co.lalev.javabook;
2
3 class Test {
4     private int priv;
5     public int pub;
6     int def;
7 }
8
9 public class Main extends Test {
10
11     public void testTheTest() {
12         priv=5; //Директен достъп. private поле. Грешка!
13         pub=6; //Директен достъп. public поле. OK!
14         def=7; //Директен достъп. default поле. OK!
15     }
16
17     public static void main(String[] args) {
18         Test test = new Test();
19         test.priv=5; //Чрез променлива. private поле. Грешка!
20         test.pub=6; //Чрез променлива. public поле. OK!
21         test.def=7; //Чрез променлива. default поле. OK!
22     }
23 }

```

Разпечатка 8.18 Ефекти от досега разгледаните модификатори при директен достъп срещу достъп чрез използване на променлива

Не така стоят нещата с последния модификатор, а именно – **protected**. Този модификатор има доста особености, което именно е причината да го разгледаме последен.

Когато става въпрос за *директен достъп*, правилата за методите и полетата, маркирани с **protected** са доста праволинейни. **Полета или методи, които са маркирани **protected**, могат да бъдат достъпвани директно само от класовете в същия клас, както и от класовете-наследници на текущия клас без значение в кой пакет се намират те.**

Долната разпечатка демонстрира директен достъп до **protected** методи и полета:


```

1 package uk.co.lalev.javabook;
2
3 class Parent {
4     protected int v;
5     protected void w() { }
6 }
7
8 public class Main extends Parent {
9     public void test() {
10         v=5; //OK
11         w(); //OK
12     }
13
14     public static void main(String[] args) { }
15 }

```

Разпечатка 8.19 Директен достъп до макирани с `protected` методи и полета

На разпечатка 8.19 класът `Main` е наследник на `Parent`, което му позволява да достъпва полетата и методите директно. Модификаторът `protected` позволява директния достъп, поради което декларациите на редове 10 и 11 се компилират успешно.

Правилата за достъп през променлива са малко по-особени. **Когато се използва променлива за достъп до класа, променливите и методите, на класа, маркирани с `protected`, могат да се достъпват от същия клас и от класовете в същия пакет.** Това е демонстрирано на следващата разпечатка:

```

1 package uk.co.lalev.javabook;
2
3 class Test {
4     protected int v;
5     protected void w() {};
6 }
7
8 public class Main {
9
10     public static void main(String[] args) {
11         Test test = new Test();
12         test.v = 5; //OK. Намираме се в същия пакет като този на Test.
13         test.w(); //OK. Намираме се в същия пакет като този на Test.
14     }
15 }

```

Разпечатка 8.20 Достъп до макирани с `protected` методи и полета чрез променлива

В допълнение, ако имаме два класа в отношение родител – наследник, без значение в кой пакет се намират те, съществува възможност да извикваме **protected** методите и полетата на класа родител от методите на класа-наследник през променлива. За целта променливата трябва да е от типа на *класа-наследник* или клас, който е негов наследник надолу по обектната йерархия.

Remote.java	
1	package uk.co.lalev.javabook.test;
2	
3	public class Remote {
4	protected int v;
5	protected void w() { }
6	}
Main.java	
1	package uk.co.lalev.javabook;
2	
3	import uk.co.lalev.javabook.test.Next;
4	import uk.co.lalev.javabook.test.Remote;
5	
6	public class Main extends Remote {
7	public void test() {
8	v=5; //OK.Директен достъп от наследник към protected поле на родител
9	w(); //OK.Директен достъп от наследник към protected поле на родител
10	}
11	
12	public static void main(String[] args) {
13	Remote p = new Remote();
14	p.v=5; //Грешка!
15	
16	Main main = new Main();
17	main.v=5; //OK!
18	
19	Remote main1 = new Main(); //!!!! Полиморфизъм
20	main1.v=5; //Грешка
21	
22	Next next = new Next();
23	next.v=6; //OK!
24	}
25	}
Next.java	
1	package uk.co.lalev.javabook.test;
2	
3	import uk.co.lalev.javabook.Main;
4	
5	public class Next extends Main { }

Разпечатка 8.21 Достъп до макрирани с **protected** методи и полета чрез променлива

Програмата в рапечатка 8.21 се състои от три класа, които изграждат обектна йерархия. `Remote` е клас-родител на `Main`, а `Main` е родител на `Next`. `Remote` и `Next` са в различен пакет от `Main`, така че правилата за достъп до `protected` полета и методи от класове в същия пакет не важат.

На редове 7 и 8 в `Main.java` е демонстрирано, че тъй като `Main` е наследник на `Remote`, неговите методи могат да достъпват `protected` методи и полета на `Remote` директно.

Много по интересен случай е демонстриран на ред 14. На този ред използваме променлива от тип `Person`, която сочи към обект от същия тип. Чрез променливата правим опит да достъпим `protected` полетата и методите на `Person`, но това е неуспешно, тъй като не е позволено в Java, дори при положението в примера, при което `Main` е наследник на `Person`. Правилото, формулирано по-горе, казва друго – възможно е да достъпим `protected` методите и полетата на `Person` от метод във `Main` чрез променлива, но ако става въпрос за променлива от тип `Main` или тип-наследник на `Main`.

Именно това е демонстрирано на редове 16 и 17 в `Main.java`. На ред 16 създаваме променлива от тип `Main`, която сочи към новосъздаден обект от същия тип. На ред 17 използваме тази променлива, за да достъпим `protected` поле, наследено от `Remote`.

Читателите може би забелязаха, че в рамките на последната страница сменихме формулировката и започнахме да уточняваме вида на обектите, към които сочат променливите от нашите променливи от референтен тип. Това е така, защото постепенно се приближаваме към последния механизъм на обектно-ориентираното програмиране, за който старателно избягвахме да говорим до момента. Благодарение на полиморфизма, който ще разгледаме след малко, можем да имаме и ситуации, при които променлива от типа-родител съдържа обект от типа-наследник. Тъй като типът наследник има всички полета и методи на родителя (макар и евентуално предефинирани), той може да замести типа-родител навсякъде където се изисква това.

Ситуацията на ред 19 е такава. Създаваме обект от тип `Main`, но го присвояваме на променлива от тип `Remote`. Правим това, за да определим дали променливата или типа на самия обект определят това дали ще имаме достъп до `protected` методите и полетата, наследени от `Remote`.

Както се вижда от ред 20, променливата определя правата за достъп, а не конкретния указван обект. Ето защо имаме ситуация, аналогична на тази на ред 14 – грешка при компилация.

Последният експеримент касае използването на променлива от тип клас-наследник на `Main`. На редове 22 и 23 това е класът `Next`. Както е видно от примера, достъпът до `protected` методите и полета, наследени от `Parent` и след това от `Main`, е позволен по този начин.

За програмистите, които се опитват да „наредят“ в главата си правилата за достъп до `protected` полетата е полезно да направим обобщение. Полетата и методите, маркирани `protected`, основно са предназначени за директно използване от класовете-наследници. Но има определени операции, които налагат изключения. Такива операции изискват метод в класа да работи върху инстанции на същия клас или инстанции на класовете-наследници. Такава операция например би била копирането на обект в друг от същия тип. Това предполага и изключението, което бе демонстрирано току-що.

Долната таблица обобщава ефектите от модификаторите за видимост при *директен достъп*:

	достъп от същия клас	достъп от клас в същия пакет /наследник или не/	достъп от клас в друг пакет, който е наследник	достъп от клас в друг пакет, който не е наследник
<code>public</code>	да	да	да	да
<code>protected</code>	да	да	да	не
(по подразбиране)	да	да	не	не
<code>private</code>	да	не	не	не

Табл. 8.2. Модификатори за достъп до методи и полета на клас в Java при директен достъп

Следващата таблица пък обобщава ефектите от модификаторите за видимост при достъп чрез променлива:

	достъп от същия клас	достъп от клас в същия пакет /наследник или не/	достъп от клас в друг пакет, който е наследник; променливата е от типа на класа наследник или под него в обектната йерархия	достъп от клас в друг пакет, който е наследник; променливата е от типа на наследявания клас	достъп от клас в друг пакет, който не е наследник
<code>public</code>	да	да	да	да	да
<code>protected</code>	да	да	да	не	не
(по подразбиране)	да	да	не	не	не
<code>private</code>	да	не	не	не	не

Табл. 8.3. Модификатори за достъп до методи и полета на клас в Java при достъп чрез променлива

Завършваме настоящата точка с наблюдението, че модификаторите за достъп могат да се прилагат и върху цели *класове*. Ние видяхме множество примери за това в разпечатките досега. Време е да споменем обаче, че имаме два вида класове – такива, които са поместени директно в пакет, и такива, които са *вложени в други класове или методи*. Досега не сме се срещали с втория вид класове, но е редно да кажем, че те могат да получават всички споменати модификатори за достъп. Ние ще разгледаме ефектите от тези модификатори когато разгледаме вложените класове по-нататък.

Всички класове, които наблюдавахме до момента, са директно поместени в пакети. Ние наричаме такива класове „**класове от най-горно ниво**“. Това понятие няма нищо общо с обектната йерархия на наследяването, а касае къде е поместен този клас. Класовете от най-горно ниво могат да получават само два модификатора – `public` и „по подразбиране“.

Класовете от най-горно ниво, маркирани като **public**, могат да се ползват от класове извън пакета. Класовете, маркирани като **private**, могат да се използват само от същия пакет. Долният пример демонстрира тези разлики:

Accessible.java	
1	package uk.co.lalev.javabook.test;
2	
3	class Internal { }
4	
5	public class Accessible { }
6	
7	public class Extra { } //Грешка при компилация
Main.java	
1	package uk.co.lalev.javabook;
2	
3	import uk.co.lalev.javabook.test.Accessible;
4	import uk.co.lalev.javabook.test.Internal; //Грешка!
5	
6	public class Main extends Internal { //!!!
7	public static void main(String[] args) {
8	Accessible a = new Accessible();
9	}
10	}

Разпечатка 8.22 Модификатори за достъп, приложени към клас от най-горно ниво

Програмата на разпечатка 8.22 е съставена от два файла, които поместват няколко класа с различни модификатори за достъп. Програмата генерира целенасочено няколко грешки, за да демонстрира казаното по-горе. Първата грешка се намира на ред 7 в `Accessible.java`. Дори без да дискутираме ефектите на модификаторите към класовете, много пъти вече споменахме, че класове от най-горно ниво, маркирани като **public**, трябва да се намират в отделен файл, който има същото име като класа и разширение `.java`. В случая декларацията на клас `Extra` води до грешка именно по тази причина.

Към класовете с разрешения за достъп по подразбиране няма такива ограничения. Те могат да бъдат дефинирани сами във файл или пък (както е в случая) да бъдат дефинирани във файловете на други класове.

Другата грешка е на ред 4 в Main.java. Грешката е породена от опит за импортиране на клас, който има видимост по подразбиране и се намира във външен пакет.

4. Използване на **super** и **this**

Наследяването създава ситуации, в които имената на методите и полетата в класа-наследник могат да съвпадат с тези на класа-родител. Предефинирането на методи е пример как такова съвпадение е целенасочено позволено в спецификацията на езика.

Когато се нуждаем от достъп до поле на класа-родител, което има същото име като поле на настоящия клас, както и в случай, когато имаме предефиниран метод и трябва да извикаме версията на метода от класа-родител, ние използваме ключовата дума **super**.

Долната разпечатка демонстрира използването на **super**, както за достъпване на полета, така и за извикване на методи на класа родител.

```
1 package uk.co.lalev.javabook;
2
3 class Parent {
4     int a=5;
5     void b() { System.out.println("Привет от Parent.b()!"); }
6 }
7
8 class Child extends Parent {
9     int a=6;
10    @Override
11    void b() {
12        super.b();
13        System.out.println("Привет от Child.b()!");
14        System.out.println("а на Parent е "+super.a);
15        System.out.println("а е "+a);
16    }
17 }
18
19 public class Main {
20
21     public static void main(String[] args) {
22         Child c = new Child();
23         c.b();
24     }
25 }
```

Разпечатка 8.23 Използване на ключовата дума `super`

И двата класа имат поле с едно и също име, както и метод, който е предефиниран в `Child`. Ред 12 демонстрира сравнително често срещано явление при предефинираните методи. Вместо да заменя напълно функционалността на метода в класа-родител, предефинираният метод го извиква като част от собственото си изпълнение, което включва и допълнителни декларации. За целта той просто префиксира името на метода с ключовата дума `super`.

На ред 13 същият подход е използван за осъществяване на достъп до полето `a` на `Parent` от метода `b` на `Child`.

Изпълнението на цялата програма извежда следния текст:

```
Привет от Parent.b()!  
Привет от Child.b()! а на Parent е 5
```

Възможността даден метод да има параметри, чиито имена съвпадат с имената на полета на класа, налага съществуването на подобен на `super` механизъм, който да се използва за обозначаване на *текущия* клас. Този механизъм е достъпен чрез ключовата дума `this`. Употребата на тази ключова дума е демонстрирана на следващата разпечатка:

```
1 class Pair {  
2     private int a;  
3     private int b;  
4  
5     public void setA(int a) { this.a = a; }  
6     public void setB(int b) { this.b = b; }  
7 }
```

Разпечатка 8.24 Използване на ключовата дума `this`

Програмата на разпечатката демонстрира типична ситуация, в която даден клас използва гетери и сетери. От съображения за краткост обаче, гетерите са

изпуснати. Типично за сетерите е техните параметри да имат същите имена като полетата, което създава конфликт. **За щастие, ключовата дума `this` позволява да уточним, че не се интересуваме от параметъра с дадено име, а от полето на текущия обект.** Употреба в този смисъл е показана на редове 5 и 6 на разпечатката. Там `this.a` например обозначава полето на обекта, докато `a` е параметър. Тоест ред 5 присвоява параметъра на полето, което е идеята на целия метод. Читателите следва да сравнят тази разпечатка, която показва типичния подход към писане на гетери и сетери, с разпечатка 8.15, където сме избрали друг подход, за да отложим дисусията на `this`.

Ключовата дума `this` може да се използва и самостоятелно. В такъв случай тя се интерпретира като референция към текущия обект. Долната разпечатка илюстрира този начин на употреба на `this`.

```
1  class Pair {
2      int a;
3      int b;
4
5      public static void swap(Pair v) {
6          int c=v.a;
7          v.a=v.b;
8          v.b=c;
9      }
10
11     public void swap() {
12         swap(this);
13     }
14 }
15
16 public class Main {
17     public static void main(String[] args) {
18         Pair p = new Pair();
19         p.a = 5;
20         p.b = 6;
21         p.swap();
22     }
23 }
```

Разпечатка 8.25 Използване на ключовата дума `this` като референция към текущия обект

Горната разпечатка демонстрира модификация на класа `Pair` с два овърлоуднати метода с име `swap`. Първият извършва размяна на стойностите на двете полета в обект от тип `Pair`, който е подаден като параметър. Вторият

метод извършва промяна текущия обект. За да спести повторението на код, вторият метод просто извиква първия. Това обаче поражда проблем. Първият метод изисква референция към обект.

Именно такава референция се съдържа в променливата `p` в `Main`. Всъщност именно тази променлива е използвана за извикването на метода `swap()` на ред 21. Проблемът е, че извикваният метод (тоест втората версия на метода, която не взема параметри) не получава копие на `p`. Следователно методът няма референция към обекта, върху който е извикан. Начинът да се получи тази референция вътре в метода е именно ключовата дума `this`, както е демонстрирано на ред 12.

Долната разпечатка демонстрира още веднъж важния факт, че `this` винаги сочи към текущия обект.

```
1 package uk.co.lalev.javabook;
2
3 class Test {
4     int a;
5     Test getReference() {
6         System.out.print("a is "+a+" ");
7         return this;
8     };
9 }
10
11 public class Main {
12     public static void main(String[] args) {
13         Test test1 = new Test();
14         test1.a = 5;
15         Test test2 = new Test();
16         test2.a = 6;
17
18         System.out.println(test1==test1.getReference()); //a is 5 true
19         System.out.println(test2==test2.getReference()); //a is 6 true
20     }
21 }
```

Разпечатка 8.26 `this` винаги съдържа референция към текущия обект

В разпечатката променливите `test1` и `test2` са инстанции на класа `Test`, а методът `getReference` е замислен така, че да връща референция към текущия обект, която може да бъде проверена за равенство.

Когато променливата `test1` се използва за извикване на `getReference`, `this` сочи към същия обект като `test1`, тъй като той е „текущият“ обект. Второто извикване на ред 19 използва `test2`. Тъй като обектът, сочен от `test2`, сега е текущ, `this` връща референция към него.

Завършваме с наблюдението, че `super` не може да се използва самостоятелно, както видяхме да се използва `this`. Когато се създава обект от даден клас, механизмите за наследяване копират и предефинират методите на класовете-родители нагоре по йерархията, както бе илюстрирано. Но в крайна сметка се създава само един обект от желания тип. Казано по-друг начин създаването на обект *не създава* и обект от родителския клас. Ето защо ключовата дума `super` няма смисъл използвана самостоятелно.

Освен очертаните начини на употреба, `this` и `super` могат да се извикат и със синтаксис, подобен на този при извикването на методи. В този случай след една от двете ключови думи се изписват параметри, заградени в скоби. По този начин се извикват методи-конструктори на текущия и родителския клас – нещо което ще разгледаме в следващата точка.

5. Конструктори

Връщайки читателите към разпечатка 8.15, която демонстрираше използване на гетери и сетери, ние ще зададем нов въпрос. Маркирането на полетата като `private` и използването на гетери и сетери дава гаранции, че индивидуалното поле ще се инициализира винаги с проверени за коректност данни. Какво ще се случи обаче, ако програмистът *въобще пропусне* да постави стойност в някое поле със задължителен характер? Такова изпускане всъщност е направено на разпечатка 8.15, когато програмистът инициализира `person` с възраст, но не и име.

Методите-конструктори са начинът за противодействие на този проблем при обектно-ориентираното програмиране. Те имат за задача инициализирането на

полетата на новосъздадения обект с валидни стойности, така че към момента на приключване на метода, обектът да е валиден и готов за употреба. За да се предотврати възможността програмистът да създаде обекта и после да забрави да извика конструктора, в Java и всички модерни програмни езици конструкторите се извикват едновременно със създаването на обекта.

Конструкторите са особени методи, тъй като трябва да работят добре с ключовата дума `new`. Това изискване налага две важни особености на конструкторите. **Конструкторите не могат да връщат стойност и техните имена трябва да съвпадат с името на класа, който ще бъде създаван.**

Долният пример демонстрира използването на конструктор:

```
1 package uk.co.lalev.javabook;
2
3 class Person {
4     private String name;
5     private int age;
6
7     public Person(String name, int age) {
8         this.name = name;
9         this.age = age;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public int getAge() {
17        return age;
18    }
19 }
20
21 public class Main {
22     public static void main(String[] args) {
23         Person person = new Person("Иван Петров Ангелов", 24);
24         System.out.println(person.getAge()+" "+person.getName());
25     }
26 }
```

Разпечатка 8.27 Дефиниране на метод-конструктор

В разпечатка 8.27 конструкторът се намира на редове 7-10. Докато методите могат да изберат между това да връщат и да не връщат стойност (за последното те сигнализират с ключовата дума `void`), конструкторите никога не връщат

стойност, поради което не се нуждаят от позиция за нея в дефиницията си. Ето защо декларираният от нас конструктор има само модификатор за видимост. Примерът от разпечатката демонстрира и една стандартна практика - параметрите на конструктора имат същите имена като тези на полетата в класа. Вътре в конструктора тези имена влизат в конфликт, поради което имената на полетата трябва да се префиксират с ключовата дума `this`.

На ред 23 в разпечатката читателите също трябва да забележат как конструкторът се извиква заедно със създаването на обекта. За целта след ключовата дума `new` се изписва името на класа/конструктора (те са винаги еднакви в Java) като в скобите се поставя списъкът с параметри.

Важен за отбелязване факт е и това, че наличието на конструктори позволява създаването на *неизменчиви*¹ обекти. Полетата на такива обекти се инициализират при създаване с конструктора. Тъй като класовете на тези обекти имат гетери, но не и сетери и тъй като методите на обекта не променят данните в полетата, съдържанието на полетата не може да се променя по времето на живота на обекта. Често алгоритмите за работата с изменчиви обекти могат да се преформулират в алгоритми за работа с неизменчиви обекти. Резултатът от този процес е това, че обработката на данните се свежда до трансформиране на един неизменчив обект в друг такъв неизменчив обект, създаден на база данните от първия. Този начин на работа носи предимства, когато например извършваме високопроизводителна многопоточна обработка на данните, ангажираща всички ядра на процесора на компютъра.

Конструкторите в Java са задължителни. Ако програмистът не използва конструктор, Java ще създаде такъв при компилация на класа. Създаденият конструктор няма да извършва никакви операции с полетата на обекта и няма да има параметри. На този етап читателите вероятно осъзнават, че в примерите досга многократно извиквахме такива конструктори по

¹ От англ. „immutable”.

подразбиране. Това се случваше всеки път, когато използвахме ключовата дума `new` за създаване на обект, за който не бяхме задали конструктор.

Тук е мястото да отбележим, че използването на конструктор е безспорна добра практика и читателите винаги трябва да създават конструктори за своите класове. Читателите също скоро ще забележат, че професионално изготвените класове, като тези в стандартната библиотека на Java, често използват множество овърлоуднати конструктори. Тези конструктори улесняват програмистите при създаването на по-сложни класове.

Когато създаването на класа е още по-сложно (може би заради отношението му към другите класове в обектната йерархия), създателите му могат да изберат вариант при който ползвателите на класа *не* го създават директно чрез извикване на ключовата дума `new` и конструктор. Вместо това те могат да викат специално създадени за целта *методи-фабрики* и *класове-фабрики*. Така например, когато за създаването на обекта имаме множество *опционални* параметри, създателите на обекта могат да изберат той да се създава чрез наречените „билдер-класове“¹.

Споменатите току-що понятия са просто креативни начини за използване на вече дискутираните механизми на обектно-ориентираното програмиране и попадат в предметната област на софтуерното инженерство, поради което няма да бъдат обяснени в детайли в настоящата книга². Читателите обаче трябва да знаят, че тези „шаблони на дизайн“ при конструирането на обекти не отменят нуждата от конструктори, но често налагат тези конструктори да бъдат маркирани като `private` или `protected` за да се предотврати директното им извикване от ползвателите на класовете чрез използването на `new`.

¹ От англ. „builder” – „строител”.

² Авторът препоръчва две книги в тази област. За „академично“ изложение – Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. За по-неформално – Freeman, Eric, *Head First Design Patterns: A Brain-Friendly Guide*. O'Reilly Media, 2004

Важно свойство на конструкторите, с което всеки програмист на Java трябва да бъде запознат, е това че при наследяване на класове, конструкторът на класа родител *винаги* трябва да бъде извикан.

Ако програмистът пропусне да направи това действие, Java ще го извърши вместо него. В комбинация с това, че конструкторите нямат собствени имена, както и това, че един клас може да има множество конструктори, които да се извикват един – друг, това създава множество усложнения и особености, които трябва да бъдат обяснени.

Долният програмен фрагмент демонстрира как един конструктор на класа може да извика друг конструктор, както и как конструкторът на класа-наследник извиква експлицитно конструктора на класа-родител.

```
8  class Person {
9      private String name;
10     private int age;
11     private String country;
12
13     public Person(String name, int age, String country) {
14         this.name = name;
15         this.age = age;
16         this.country = country;
17     }
18
19     public Person(String name, int age) {
20         this(name, age, "Bulgaria");
21     }
22 }
23
24 class Client extends Person {
25     private String clientNo;
26     public Client(String clientNo, String name, int age, String country) {
27         super(name, age, country);
28         this.clientNo=clientNo;
29     }
30
31     public Client(String clientNo, String name, int age) {
32         this(clientNo, name, age, "Bulgaria");
33     }
34 }
```

Разпечатка 8.28 super() и this()

Програмният сегмент на разпечатка 8.28 съдържа два класа, всеки от които има два конструктора. Вторият конструктор и на двата класа няма параметър

country. Идеята е когато се извика този конструктор, той да попълни най-често използваната стойност - "Bulgaria" в съответното поле.

За да се избегне повторението на код при конструкторите в подобни сценарии, Java позволява един конструктор да извиква друг конструктор. Това става чрез ключовата дума `this`, следвана от скоби и списък с параметри, които ще бъдат предадени на конструктора. Такова извикване е демонстрирано на редове 20 и 32 на разпечатката.

Подобно на `this`, използването на `super` със скоби и списък от параметри извиква конструктора на родителския клас. Тъй като `Client` е клас-наследник на `Person`, той на някакъв етап задължително трябва да извика конструктора на `Person`. Това се случва на ред 27 в примера.

Върху примера от разпечатката могат да се направят много наблюдения, които на свой ред пораждаят още въпроси. Както е видно от разпечатката, не е задължително *всеки* конструктор да извиква конструктора на класа-родител. Конструкторът на класа-родител трябва да бъде извикан на някакъв етап от поне един от конструкторите на класа-наследник, но съществува и възможност даден конструктор да извика *друг* конструктор на същия клас. Дали обаче това са единствените две възможности?

Оказва се, че това е така. **Всеки конструктор в Java трябва да започва с едно от двете – извикване на друг конструктор на същия клас чрез `this`, или извикването на конструктора на класа-родител чрез `super`. Ако такова извикване липсва, Java ще добави автоматично `super` като първа декларация.**

Java обаче може да извиква само конструктори без аргументи, тъй като очевидно няма как да знае какви аргументи би задал програмиста. Ако родителският клас няма конструктор без аргументи, а конструкторът на класа-наследник не започва с извикване на `super` или `this` с нужните параметри, това ще доведе до грешка при компилация. Също така трябва да се помни, че

извикването на `super` и `this` /експлицитно или имплицитно, вмъкнато от Java/ може да се прави само веднъж – точно като първа декларация на конструктора.

Долният пример илюстрира някои грешки, породени от нарушаване на горните правила:

```
8  class Parent {
9      public Parent() {
10         System.out.println("Hello from Parent!");
11     }
12 }
13
14 class Child extends Parent {
15     public Child(int a, int b) {
16         //Имплицитно извикване на super()
17         System.out.println("Hello from Child!");
18     }
19 }
20
21 class GrandChild extends Child { } //Грешка
22
23 class AnotherGrandChild extends Child {
24     public AnotherGrandChild() { //Грешка
25         System.out.println("Hello from AnotherGrandChild!");
26     }
27 }
28
29 class First { }
30
31 class Second extends First {
32     public Second() {
33         System.out.println("Hello from AnotherGrandChild!");
34         super(); //Грешка
35         this();  //Грешка
36     }
37 }
```

Разпечатка 8.29 Грешки при извикване на `super()` и `this()`

В програмата на разпечатка 8.29, класът `Parent` има конструктор, зададен от програмиста. Това означава, че Java няма да създаде *конструктор по подразбиране*. Създаденият от програмиста конструктор в случая е без аргументи, така че ще може да бъде извикван автоматично от конструкторите на класовете-наследници.

Ние ще проверим това правило в класа `Child`, който наследява `Parent`. `Child` има един конструктор с два аргумента. Тъй като този конструктор не започва със `super` или `this`, Java ще вмъкне `super` автоматично. В случая това

работи, тъй като класът-родител има конструктор без аргументи. Ако не беше така, конструкторът `Child` на ред 15 щеше да генерира грешка при компилация.

Класът `Child` има конструктор, поради което, разбира се, Java няма да създаде конструктор по подразбиране, но когато Java създаде класа `GrandChild`, който няма конструктор, такъв най-накрая в нашия пример ще бъде създаден. Конструкторът по подразбиране на `GrandChild` ще опита да направи само едно - ще опита да извика този конструктор на `Child`, който е без аргументи. Такъв няма и това ще породи грешка при компилация на ред 21.

При `AnotherGrandChild` програмистът е създал конструктор, но без референция към `this` или `super`. При това положение Java ще опита да вмъкне извикване на конструктора без аргументи на `Child`. Това ще генерира аналогична на предната грешка.

Грешката на ред 35 пък се дължи на това, че `super` не е първата декларация в конструктора. Ред 36 задълбочава предната грешка в две посоки. Първо, в един конструктор може да има или `this` или `super`, но не и двете извиквания. Второ, ако `this` трябваше да бъде извикана, тази декларация трябваше да бъде първа в конструктора.

Тук е моментът да се отбележи също, че *рекурсивните* извиквания (тоест ситуациите, в които един метод извиква себе си) са забранени при конструкторите, независимо дали рекурсията се случва директно или става въпрос за „извикване в цикъл“, при което изпълнението тръгва от един конструктор и се връща към него след определен брой извиквания на други конструктори с `this`. Така че дори `this` да беше първото и единствено извикване на друг конструктор в `Second`, тъй като списъкът параметри води до това, че конструкторът извиква себе си, това щеше да бъде грешка.

Класовете в Java винаги се дефинират в *пакети*, което също е форма на инкапсулация. Пакетите съдържат набор от класове, които са създадени за извършване на някаква конкретна задача и са по някакъв начин свързани.

Така например класът, който представлява един студент, има множество информационни атрибути, като ЕГН, факултетен номер, специалност и т.н. Някои от тези атрибути обаче също са „съставни“ и трябва да се дефинират като отделни класове. Такъв атрибут например е оценката, която „носи в себе си“ числова оценка и оценка с думи, както и предмет, по който е написана. Самият предмет също може да е обект, тъй като той също има множество информационни атрибути, като например наименование, брой часове в учебната програма и т.н.

Класовете от дадения пример не могат да функционират един без друг, което означава, че те ще бъдат поставени в един пакет. Малките програми обикновено се състоят от един пакет, докато големите програми могат да са съставени от повече пакети.

Как задаваме в кой пакет бива поставен даден клас? В началото на всеки *файл* следва да се постави декларация `package`, която се състои от ключовата дума `package` и уникално име на пакета, избрано от програмиста. Всички класове, които са дефинирани в този файл се считат за числящи се към този пакет.

Читателите могат да видят такава декларация на ред 1 на разпечатка 8.2, както и в началото на повечето примери в настоящата книга. Важно е да се каже, че **освен коментарите, които впрочем не се считат за декларации в езика Java, никоя декларация не може да *предхожда* декларацията `package`. Тя трябва да бъде първата декларация във всеки файл.**

Ако декларацията `package` е изпусната, Java ще постави класовете от файла в пакета по подразбиране. Тези класове *не могат* да се споделят и да бъдат използвани от други пакети.

В Java има стриктни правила за имената на пакетите. Те могат да се състоят от части, разделени помежду си с точки. Всяка от частите трябва да е валиден

идентификатор в Java. В рамките на една програма не са позволени пакети с еднакви имена и за разлика от имената на класовете, имената на пакетите в Java трябва да се изписват само с малки букви.

Програмстите често споделят програмен код помежду си, като най-малката единица за споделяне е именно пакетът от класове. Споделянето на пакети създава проблем с имената. Какво ако двама програмиста нарекат пакетите си по един и същи начин, след което тези два пакета, по стечение на обстоятелствата станат част от една програма? Това, както казахме току-що, не е позволено в Java.

За да решат този проблем, програмистите на Java по традиция създават на *глобално-уникални* имена на пакетите. Тези имена не се повтарят в целия свят. За да се получи такова име се използва комбинация от домейна на организацията в Интернет, изписан „наобратно“, и името на проекта. Така например организация, която има домейн `google.com` и проект, който се нарича `Guava`, може да нарече пакета на този проект `com.google.guava`. Ако пък проектът се състои от множество пакети, всички техни имена ще започват с `com.google.guava`, и ще продължават с различни идентификатори за отделните пакети.

Тъй като системата с домейн имена позволява някои символи и имена, които не са валидни идентификатори в Java, програмистите заместват тиретата (валидни в системата с домейн имена, но невалидни при Java идентификаторите) с подчертаваща черта. Те също префиксират домейните, започващи с цифри (невалидни идентификатори в Java) с подчертаваща черта (вж. табл. 8.1).

Домейн	Примерни имена на пакет
123start.com	com._123start.package1 com._123start.package2
uni-sofia.bg	bg.uni_sofia.testpackage bg.uni_sofia.somename

Табл. 8.1. Имена на пакети от домейн имена, които съдържат невалидни идентификатори в Java

Формираните по указания начин имена се използват за всички пакети с едно важно изключение - пакетите от стандартната библиотека на Java и нейните разширения имат имена, които започват съответно с `java` и `javax`. Така например в стандартната библиотека на Java имаме пакети с имена `java.util` и `java.lang`, както и `javax.net` и `javax.crypto`. Тези имена не кореспондират на домейн имена, но не могат да влязат в конфликт с имената, съставени по гореописаното правило, тъй като в Интернет няма (и няма да има) домейн имена, завършващи на `.java` и `.javax`.

За да използваме един клас от друг пакет, ние трябва да изпишем неговото „**напълно квалифицирано име**“, което се състои от името на пакета, комбинирано с името на класа. Така например напълно квалифицираното име на класа `Date` от пакета `java.util` е `java.util.Date`. Долният пример демонстрира използването на класове по този начин:

```
1 package uk.co.lalev.javabook;
2
3 public class Main {
4     public static void main(String... args) {
5         java.util.Date date;
6         java.util.Date generator = new java.util.Date();
7
8         java.sql.Date sqlDate;
9     }
10 }
```

Разпечатка 8.3 Използване на класове от външни пакети

На разп. 8.3 класът `Main` се намира в пакета `uk.co.lalev.javabook`, а класът `Date` се намира в `java.util`. Това значи, че класът `Date` е извън пакета на класа `Main`.

Следователно, когато трябва да използваме `Date` в `Main`, ние трябва да използваме напълно квалифицираното име на този клас. Редове 5 и 6 демонстрират дефинирането на две променливи от тип `java.util.Date`. Първата от тях е неинициализирана и не сочи към създаден обект, докато втората е инициализирана и сочи към новосъздаден обект в хийпа на програмата.

Читателите вероятно са съгласни, че използването на класове по този начин е неудобно, тъй като техните напълно квалифицирани имена са дълги и се изписват бавно. Въпреки това на пръв поглед те изглеждат като необходимо зло, тъй като кратките имена могат да се дублират. Така например ред 8 в примера демонстрира, че в стандартната библиотека с класове на Java има и друг клас, който се нарича `Date`. Той се намира в пакета `java.sql` и неговото напълно квалифицирано име е `java.sql.Date`. Ако използвахме кратките имена на класовете, нямаше да е ясно кой от двата класа имаме предвид.

За щастие, рядко се случва да има нужда да се използват два външни за пакета класа с едно и също име в един *файл* с изходен код. По същия начин не са чести ситуациите, когато краткото име на използван външен за пакета клас съвпада с едно от имената на класовете в текущия *файл*. Поради тази причина, **когато няма съвпадения на имената, почти винаги програмистите на Java използват декларацията `import`. Тя която позволява външен клас да се използва с краткото си име в текущия файл.**

Долният пример представлява редакция на примера от 8.3, така че да се използва `import`.

```
1 package uk.co.lalev.javabook;
2
3 import java.util.Date;
4
5 public class Main {
6     public static void main(String... args) {
7         Date date;
8         Date generator = new Date();
9
10        java.sql.Date sqlDate;
11    }
12 }
```

Разпечатка 8.4 Редакция на пример 8.3 с използване на `import`

На ред 3 в примера класът `java.util.Date` е импортиран, поради което на редове 7 и 8 се използва с краткото си име, точно както би се използвал, ако беше дефиниран в текущия пакет.

За съжаление не е възможно и двата класа с кратко име `Date` да бъдат импортирани, както и не е възможно да бъде импортиран клас, чието кратко име съвпада с клас от текущия файл. Долната разпечатка илюстрира тази ситуация с пример:

```
1 package uk.co.lalev.javabook;
2
3 import java.util.Date;
4 import java.sql.Date; //Грешка при компилация
5
6 import java.util.Random;
7
8 class Random {           //Грешка при компилация
9
10 }
11
12 public class Main {
13     public static void main(String... args) { }
14 }
```

Разпечатка 8.5 Импортирането на класове с еднакви кратки имена не е позволено

Примерът на разпечатка 8.15 дори не използва никой от импортираните класове, въпреки това импортирането предизвиква грешки. Първата от тях е на ред 4, където компилаторът ще забележи, че това е *вторият* клас с кратко име `Date`, който импортираме. Тъй като това създава условия за двусмислие по-нататък, компилаторът ще издаде съобщение за грешка още на този ред.

Импортирането на класа `java.util.Random` минава успешно, но когато се опитаме да дефинираме клас, който има същото кратко име, това е грешка, тъй като сега имаме два класа с кратки имена `Random`.

Декларацията `import` има още форми, които също имат своите особености. Така например, декларацията `import` позволява поставянето на `*` на мястото на името на класа. Този **уайлдкард** ще доведе до това, че ще бъдат импортирани едновременно всички класове от посочения пакет. В ранните дни на Java такъв начин на импортиране бе необходим, за да се избегне писането на десетки редове `import` директиви ръчно. С появата на модерните интегрирани среди, `import`

директивите в много случаи се пишат директно от средата, което премахва нуждата от използването на уайлдкард импортиране.

Долната разпечатка илюстрира ситуации с конфликт на имената при уайлдкард импортирането:

Main.java	
1	package uk.co.lalev.javabook;
2	
3	import java.util.*;
4	import java.sql.*;
5	
6	public class Main {
7	public static void main(String... args) {
8	Date test = new Date(); //Грешка при компилация
9	}
10	}
Info.java	
1	package uk.co.lalev.javabook;
2	
3	import java.util.Date;
4	import java.sql.*;
5	
6	public class Info {
7	static void test() {
8	Date test = new Date(); //java.util.Date
9	java.sql.Date sqlDate;
10	}
11	}

Разпечатка 8.6 Конфликти на имената при импортирането чрез уайлдкард

На редове 3 и 4 в Main.java от горната разпечатка се импортират всички класове от двата пакета java.util и java.sql. Това обаче само за себе си не причинява грешка в компилацията, независимо от това, че и в двата пакета има клас с име Date. Проблемът се появява тогава, когато е направен опит да се създаде променлива чрез използване на краткото име на класа - Date. Това е двусмислено, тъй като не става ясно кой от двата класа класа се има предвид. Ето защо компилаторът ще даде грешка при тази дефиниция. Ако вместо това програмистът бе посочил напълно-квалифицираното име на класа, файлът щеше да се компилира без грешки.

На редове 3 и 4 в `Info.java` програмистът импортира специфично класа `java.util.Date` и всички класове от пакета `java.sql`. Също като при `Main.java` това само по себе си не предизвиква грешки при компилация. При това положение обаче, когато компилаторът срещне само името `Date`, той ще счете че става въпрос за по-специфично зададения клас, а именно `java.util.Date`. С други думи точно зададените класове имат приоритет пред класовете, импортирани с уайлдкард, когато трябва да бъде извършена *резолюция на имената* и да се определи кой клас от кой пакет идва. Това означава, че `Info.java`, за разлика от `Main.java`, се компилира без проблеми.

Много важен въпрос, свързан с полиморфизма, е това какво се случва с предефинираните методи, както и с полетата с еднакви имена, когато в променлива от типа на класа-родител подставим променлива от типа на класа наследник.

Всъщност, демонстрирайки полиморфизма в предните примери, ние демонстрирахме най-важния отговор – предефинираните методи на обекта-наследник се извикват вместо оригиналните методи на класа, от чийто тип е променливата.

Ние ще демонстрираме още веднъж това, добавяйки към демонстрацията и примери за някои по-специфични и редки детайли, свързани с този процес.

```
1 class Parent {  
2     boolean someField;  
3     void someMethod() {  
4         System.out.println("someMethod in Parent");  
5         System.out.println("someField is "+someField);  
6     }  
7     void someOtherMethod() {  
8         System.out.println("someOtherMethod in Parent");  
9         System.out.println("someField is "+someField);  
10 }
```

```

11 }
12
13 class Child extends Parent{
14     double someField;
15     @Override
16     void someMethod() {
17         System.out.println("someMethod in Child");
18         System.out.println("someField is "+someField);
19     }
20 }
21
22 public class Main {
23     public static void main(String[] args) {
24         Parent p = new Child();
25         Child c = new Child();
26         c.someMethod(); // someMethod in Child
27                        // someField is 0.0
28
29         p.someMethod(); // someMethod in Child
30                        // someField is 0.0
31
32         p.someOtherMethod(); //someOtherMethod in Parent
33                             //someField is false
34
35         c.someOtherMethod(); //someOtherMethod in Parent
36                             //someField is false
37
38         System.out.println(c.someField + 2.0); // 2.0
39         //System.out.println(p.someField + 2.0); // Грешка при компиляция
40     }
41 }

```

Разпечатка 8.14 Демонстрация на полиморфизъм

В примера имаме два класа в отношение родител – наследник. Това са съответно класовете **Parent** и **Child**. Тези класове имат полета, чиито имена съвпадат. Това са полетата с име **someField**. Полето **someField** в **Parent** е от тип **boolean**. И тъй като не е инициализирано чрез конструктор или по някои от другите начини, които тепърва ще бъдат дискутирани, то ще има стойност по подразбиране **false**. Полето **someField** в **Child** е от тип **double** и има стойност 0.0 по подразбиране.

Класът **Child** също така предефинира метода **someMethod** на **Parent** и наследява без промени метода **someOtherMethod** от **Parent**.

Същинската част на демонстрацията започва на редове 24 и 25, където ние декларираме променливите **p** и **c** съответно от тип **Parent** и **Child**. И двете променливи получават за стойност препратка към инстанция от тип **Child**.

Първите резултати са очаквани. На ред 26 използваме извикваме **someMethod** чрез променлива от тип **Child**. Това ще активира предефинираната версия на метода, дефинирана в **Child**. Интересно е да се отбележи, че предефинираната версия на метода използва променливата **someField** от същия клас – **Child**. Това всъщност е най-логичната възможност, тъй като методът използва с приоритет променливите от същия клас, в който е дефиниран. Това на свой ред дава максимална свобода на програмистите да променят програмната логика на класа-родител.

Резултатът от изпълнението на ред 29 разкрива естеството на връзката между референциите и самите обекти. Независимо че този път обектът от тип **Child** се достъпва през променлива от тип **Parent**, това е **Child** обект с неговите предефинирани методи, поради което извикването на **someMethod** през тази променлива в крайна сметка води до същите резултати като на ред 26.

Извикването на **someOtherMethod** на ред 32 е през променлива от тип **Parent**, докато извикването на **someOtherMethod** на ред 35 е през променлива от тип **Child**. Независимо от разликата в типа на променливите обаче, ние ги инициализирахме с обекти от тип **Child**. И в двата случая всъщност изходът на конзолата е продукт от е наследения от **Parent** и “прекопиран” без промени в **Child** програмен код на **someOtherMethod**. Ние можем да „преразкажем“ този факт и като заявим, че методите, наследени от суперкласовете на текущия клас “виждат” полетата така, както изглеждат в класа, в който е дефиниран съответния метод. Двата „мислени модела“ са еквивалентни.

За програмистите често е по-удобно да мислят за подобни наследени методи като „намиращи се“ в класа-родител. Това е възможно тъй като поведението на наследените методи по отношение на полетата е консистентно с този начин на мислене. Именно това е интересното в тази част от примера. Получените резултати на редове 32 и 35 всъщност означават, че механизмите на полиморфизма не работят по същия начин за полетата, както работят за методите.

Не е възможно методи, написани за класа-родител, да работят с полета от класове-наследници.

Последният детайл на примера касае директния достъп до поле чрез променливи от типа на класа-родител или типа на класа-наследник. На ред 38 ние демонстрираме, че когато използваме променлива от тип **Child** за да достъпим поле на обект от тип **Child** получаваме съвсем очаквания резултат – стойността на полето от **Child**. Когато на ред 39 използваме променлива от тип **Parent** за да достъпим обект от тип **Child** обаче, ние получаваме стойността на полето от **Parent**.

Това още веднъж показва, че наследените обекти пазят в себе си и полетата на класовете-родители, за да могат да ги подават на наследените методи (т.е. тези методи, които се копират без промени от класовете-родители).

Настоящият пример е доста сложен и объркан за начинаещи програмисти. Ето защо вложилите време и усилия в проумяването му читатели може би ще се възмутят от последния извод, който може би е най-важното, което трябва да бъде запомнено. Важността на споменатия факт е толкова голяма, че си заслужава „загубените“ страници и неудобството на читателите.

Преповтарянето на имена на полета в класовете-наследници със сигурност е възможно, но трябва да се избягва където е възможно като лоша практика, тъй като води до не съвсем интуитивни ситуации. Програмистите, които разширяват чужди класове, може би от „дълбоки“ обектни йерархии, не могат винаги да избират уникални имена на полетата си. Дублирането на имена в такъв случай не може да се избегне, но поведението на Java в този случай е интуитивно и „не влиза в кадър“, докато не се допускат и други лоши практики. Особено лоша такава е директния достъп до полета на обектите, както например направихме на редове 38 и 39. Полетата винаги трябва да се достъпват през методи, за да се използват в максимална степен предимствата и да се минимизират недостатъците на ООП.