Емил Хаджиколев, Станка Хаджиколева

ОСНОВИ НА ПРОГРАМИРАНЕТО С JAVA

Университетско издателство „Паисий Хилендарски”
Пловдив, 2016

Основи на програмирането с Java
Първо издание

Университетско издателство „Паисий Хилендарски“, Пловдив

ISBN 978-619-202-108-5

Пловдив, 2016

Съдържание

Предговор

Java е един от съвременните езици за програмиране. **Създаден от Sun Microsystems, с основен автор Джеймс Гослинг (James Gosling)**, сега езикът се разработва и поддържа от **Oracle**.

Java е обектно-ориентиран и много-платформен език за програмиране. Той е с отворен код, лесен за разучаване, с много стандартни библиотеки. Има множество специализирани библиотеки, създадени от различни разработчици, които също могат да бъдат ползвани безплатно.

За разлика от повечето езици за програмиране (най-популярни, от които са C, C++, C#, PHP, Python, R, JavaScript, Perl, Ruby...), които са специализирани в определени области, **Java се използва за разработката на разнообразни приложения – конзолни, с графичен потребителски интерфейс** (GUI/Graphical User Interface**), мобилни, обикновени** (стартирани на обикновен уеб сървър) и **корпоративни** (Enterprise) **уеб приложения**, **както и на приложения, управляващи** различни „умни" устройства (Интернет на нещата/Internet of Things) като **микро-контролери, сензори, телевизори** и др. Разбира се, тази всеобхватност не винаги прави Java предпочитания за конкретни разработки език.

Средите за разработка на Java, като NetBeans, Eclipse и много други, ускоряват процеса по създаване на приложенията.

Какво съдържа и за кого е предназначена книгата

Настоящата книга има за цел да въведе читателя в интересния свят на програмирането. Направен е преглед на математическите основи на програмирането. Обяснени са основните понятия в езиците за програмиране, като е акцентирано върху програмния език Java. Последователно са разгледани основните типове, величини, оператори и изрази в Java. Представени са конструкциите за контрол на изпълнението на програми. Обърнато е внимание на работата с масиви и низове. Целият материал е илюстриран с код на програмния език Java.

Книгата е създадена в резултат на многогодишно обучение по програмиране на студенти в различни бакалавърски и магистърски специалности във факултетите по Математика и информатика, Химия, Физика и Биология на Пловдивския университет „Паисий Хилендарски". Информацията в нея е представена последователно и подробно. Илюстрирана е с екрани и инструкции за работа в средата на Eclipse, поради което може да бъде използвана като самоучител от ученици и студенти, без познания за езиците за програмиране. В книгата има много решени и нерешени задачи, което я прави подходяща за обучение и самообучение на студенти. Преподавателите могат да я използват като методическо помагало.

Книгата и изходният код на примерите в нея са достъпни на адрес http://fmi.fractime.net/javabook/.

За авторите

Авторите на книгата – Емил Хаджиколев и Станка Хаджиколева са преподаватели по различни информатични дисциплини във Факултета по математика и информатика на Пловдивския университет „Паисий Хилендарски". Автор на глави 1, 3, 4, 8, 9, 10, 11 е Емил Хаджиколев, а на останалите – 2, 5, 6, 7 – Станка Хаджиколева. В текстовете на всички глави са правени взаимни допълнения, корекции и редакции.

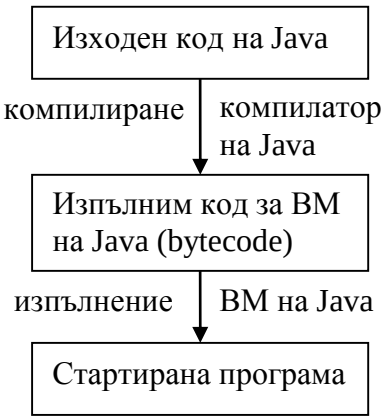
05.01.2016 г. ***Авторите***

Глава 1. Въведение в Java

Java е обектно-ориентиран език за програмиране и поддържа основните принципи на обектно-ориентираното програмиране. Кодът на програмите се записва в класове, описващи модели на различни същности (като понятия, обекти, процеси, модели и др.) от реалния свят. Класовете представят физически и функционални характеристики на тези същности, съответно чрез своите полета и методи. Конкретни представители на класовете са обектите.

Java е много-платформен език за програмиране. Базиран е на концепцията „write once, run anywhere" (в превод „пиши веднъж – изпълнявай навсякъде"). Програма, написана на Java, може да бъде стартирана на всяка платформа (хардуерна система+операционна система), която поддържа Java, например на Windows, Linux, Mac OS, Solaris; 32 или 64 битови процесори, мобилни и други устройства.

Под названието „Java" се разбира не само език за програмиране. **Java е платформа**, която освен **езика Java**, включва различни **инструменти (изпълними програми)** за различни компютърни системи, както и множество **стандартни библиотеки** от класове. **Преносимостта на Java приложенията** между отделните системи се постига като за всяка от тях се създават идентични разработки на изпълнимите програми на Java платформата.



Фиг. 1.1. Процес на компилиране и изпълнение

Изходният (source/сорс) код на една програма се компилира до междинен код, наречен „байт код" (bytecode). Този байт код може да бъде изпълнен на всяка платформа, стига да има **Виртуална машина (ВМ)** на Java (Java Virtual Machine/JVM) за съответната платформа. ВМ на Java е изпълнима програма (виртуален процесор на Java), а байткодът съдържа инструкции за него. По този начин, една програма може да бъде компилирана в една среда, а генерираният байткод да бъде изпълнен на произволна система, поддържаща Java.

Примерен клас „Hello, World!"

Тривиален примерен код за създаване на клас в Java е следният:

```
public class FirstClass {
   public static void main(String[] args) {
         System.out.println("Здравей, свят!");
   }
}
```

При изпълнение на кода, в стандартния изход (напр. в конзолата), ще се отпечата текста "Здравей, свят!".

В примера е дефиниран **клас с име FirstClass**, който притежава **метод с име main**. Описанието на клас изисква да се зададе име, предшествано от ключовата дума "class". "public" е ключова дума, която указва, че обектът, за който се отнася е общодостъпен – в случая, класът и методът също са общодостъпни. Ключовите думи и имената трябва да се изписват, без да се променя големината на буквите. Например, „public" е ключова дума, но "Public" – не е; „System" е име на стандартен клас, но „system" – не е.

Във фигурни скоби – { и } – съответно за начало и край, се описват части от код, които са свързани с клас, метод или друга обособена група от оператори. **Код, заграден във фигурни скоби се нарича блок**, а понякога и **тяло** (на клас или метод в примера).

След името на метода в обикновени скоби – (и) – се описват параметри. В примера, параметърът е масив с име args, който се състои от елементи от тип String.

След извикване на метод се записва "точка и запетая" (;), което се възприема като край на командата.

В класовете може да има дефинирани методи. **Изпълнението на дадена програма в Java започва от метод** с описаната декларация –„**public static void** main(String[] args)". Ключовата дума static означава, че метода е статичен и може да се изпълни без да се създава обект за класа, а "void" – че методът не връща стойност.

Заглавната част на метода, без неговото тяло се нарича **декларация на метода**.

Дефиницията на метод включва неговата **декларация и тяло**.

Отпечатването на текст в конзолата става с помощта на метода println на стандартния обект за изход "out", който се намира в системния клас System. Текстът, който се отпечатва, се задава като параметър на метода println. Достъпът до членовете на клас или обект става с помощта на оператор „точка" – „.".

В следващи глави ще разясним по-подробно операторите, методите, параметрите, масивите и различните ключови думи (модификатори), които определят възможните начини за връзка с други класове и методи.

Среди за изпълнение и разработка на Java приложения (JRE и JDK)

За да се изпълняват Java приложения на дадена компютърна система, е необходимо на нея да има инсталирана средата **JRE (Java Runtime Environment)**.

За създаване на Java приложения е необходимо да се инсталира комплект с **инструменти и стандартни класове за разработка JDK (Java Development Kit).**

JRE съдържа минималния набор от инструменти и библиотеки с класове за изпълнение на Java приложения. JDK предоставя всички възможности на JRE, както и допълнителни инструменти за разработка и тестване на приложения.

Инсталирането на JDK става при инсталиране на конкретна Java платформа или при инсталиране на среди за работа с Java**.**

Java платформи

Съществуват различни реализации на Java платформата, които могат да бъдат използвани при разработка на различни видове приложения:

- Java SE (Standard Edition) – стандартни библиотеки с класове и инструменти за разработка и работа с десктоп и сървърни приложения.
- Java EE (Enterprise Edition) – освен стандартните Java SE библиотеки с класове и инструменти, включва и допълнителни: за работа с уеб и приложни (application) сървъри, и свързани с тях технологии като уеб услуги, сървлети, jsp, EJB и др.;
- Java ME (Micro Edition) – използва се за създаване на приложения за мобилни устройства, сензори, микро-контролери и др., и е с ограничени, спрямо JAVA SE възможности.

Класове, пакети, API-та, JAR и ZIP архиви

Всеки клас в Java си има име.

Компилиран до байткод клас се съхранява като файл с разширение class и име като името на класа.

Обикновено, при създаването на клас се указва (в кода), че принадлежи на пакет с определено име. Пакетите може да имат под-пакети. Тогава, за да може да се използва, файлът на класа трябва да се намира в папка със същото име.

Класовете имат уникални имена в един пакет, но в различни пакети може да съществуват класове с еднакви имена. Може да се каже, че пълното име на клас (включващо пакетите),е уникално.

Множество от класове, предоставящи функционалност за определена област от дейности, се нарича API (Application Programming Interface).

Поради огромния брой на класовете и за по-лесна работа с тях, те се обединяват в архиви. Един архив може да съдържа едно или повече API-та. Архивите, които се използват в Java са jar (Java ARchive – Java технология за архивиране) и zip.

Стартиране на програма – компилиране и изпълнение

Процесът на стартиране на програма включва няколко етапа:

- Създаване на един или повече класове и записването им във файлове с подходящи имена и разширение java;
- Компилиране на класовете, при което се получават класове с разширение class;
- Стартиране на програмата.

При създаването на общодостъпен (public) клас е необходимо той да се запише във файл със същото име и разширение “java”. За разглеждания по-горе пример, кодът трябва да е записан във файла FirstClass.java.

В един java-файл може да има описани няколко класа, но само един може да е “public”.

Само ако са спазени горните правила, java-файл може да бъде компилиран.

Компилирането се извършва с помощта на компилатора на java, който е изпълним файл с име javac и се намира в bin директорията на JDK/JRE.

В следващите примерни кодове са разгледани процесите на компилиране и изпълнение под Windows. Информация за други платформи може да откриете в официалното ръководството на Java (Java Tutorials - http://docs.oracle.com/javase/tutorial/***).***

Ако нашият файл се намира в папка „C:\samples”, за да го компилираме, е необходимо да запишем в “Command Prompt” (cmd.exe) на Windows:

```
C:\samples>"C:\Program Files\Java\jdk1.8.0_66\bin\javac.exe"
FirstClass.java
```

Трябва да се отбележи, че задължително се указва пълното име на файла, заедно с разширението „.java”. В противен случай процесът на компилиране няма да е успешен.

При успешно компилиране се създава файл с име FirstClass и разширение class, който съдържа байткод.

Стартирането на програмата (файл, съдържащ main метод) става чрез извикване на ВМ на Java, на която се задава като параметър class-файла, но без разширението:

```
C:\samples>"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" FirstClass
```

Като резултат, в „Command Prompt” ще се отпечата:

```
Здравей, свят!
```

В по-стари от Windows 7 версии, „Command Prompt” показва по подразбиране само малък набор от символи (от ASCII кодовата таблица), поради което може да има проблем с визуализирането на българските символи.

Компилаторът (javac) и ВМ на Java (java) може да не се извикват с пълните им имена. Възможно е да се промени системната променлива PATH на Windows, към която да се добави път към bin директорията на JDK. Това става от „My Computer/Properties” или от „Control Panel/System” и след това Advanced/Environment Variables.

Фиг. 1.2. Компилиране и изпълнение на програма

Инструменти за програмиране на Java

Възможно е да използвате обикновен текстов редактор, като Notepad++, Wordpad и др., за да създадете програма на Java.

Съществуват **интегрирани среди за разработка** (IDE – Integrated Development Environment), които улесняват процеса по програмиране и дебъгване на приложения. Някои от средите предлагат възможности за работа с различни езици за програмиране. Различни реализации на средите за работа с Java също така може да включват възможности за работа само с определена или с всички Java платформи (CE, EE, ME).

Основни характеристики на интегрираните среди за разработка са:

- **управление на множество проекти;**
- **редактор на кода с вграден анализатор, който:**
  - с цел по-добра четимост на кода оцветява по различен начин различните елементи от кода (полета, методи, ключови думи и др.);
  - по време на кодиране предлага подходящи имена на методи и полета в зависимост от контекста;
  - предупреждава за възможна неправилна работа на кода, синтактични грешки и др.;
- **лесно стартиране на приложенията** (вградена работа с компилатора и ВМ на Java);
- **автоматизирана работа с кода** – преформатиране по определени стандартни правила за писане, автоматично добавяне на елементи и др.;
- **рефакторинг (refactoring)**, предлагащ множество възможности, сред които добавяне на специфични методи за полета на клас (set и get), преобразуване на класове и методи по избрани стандартни шаблони за дизайн и др.;
- **интеграция с документацията на Java**, предоставяща контекстна помощна информация за използваните класове и методи;
- **инструменти за дебъгване, работа в екип, готови шаблони за приложения и др.**

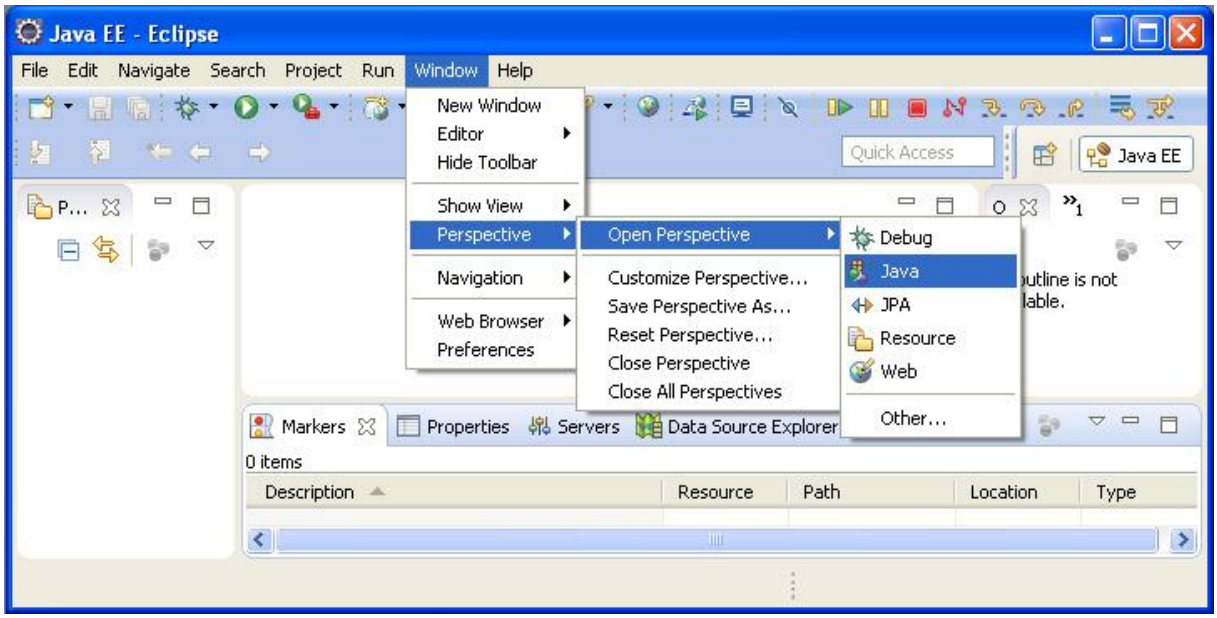
Едни от най-популярните среди с отворен код и възможност за безплатна употреба са Net Beans и Eclipse.

- Net Beans е официална среда за работа с Java. Може да се свали от сайта на NetBeans (http://netbeans.org/downloads/index.html) или заедно с JDK от сайта на Oracle. В първия случай трябва отделно да се инсталира и JDK, а във втория - не;
- Eclipse може да се свали от http://www.eclipse.org/downloads/.

За създаване на примерите на настоящата книга е предпочетена средата Eclipse, поради по-доброто представяне на кода.

Работа с Eclipse

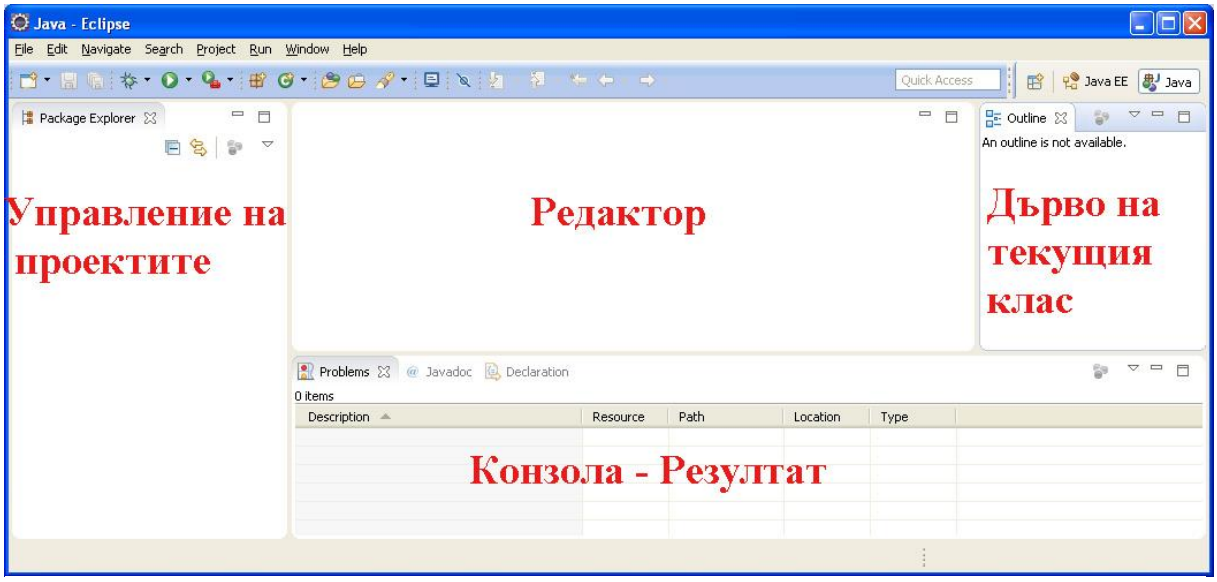
Eclipse се разпространява като архивен файл, за дадена операционна система и процесор (32 или 64 битов). Може да се избере версия за Java EE или само за Java SE. След разархивирането не е необходима допълнителна инсталация. Ако сте свалили Eclipse за Java EE, по-подразбиране е избрана перспектива „Java EE". За да работите с Java SE, трябва да смените перспективата от менюто Window/Perspective/Open Perspective/Java (фиг. 1.3).



Фиг. 1.3. Избор на Java SE перспектива в Eclipse

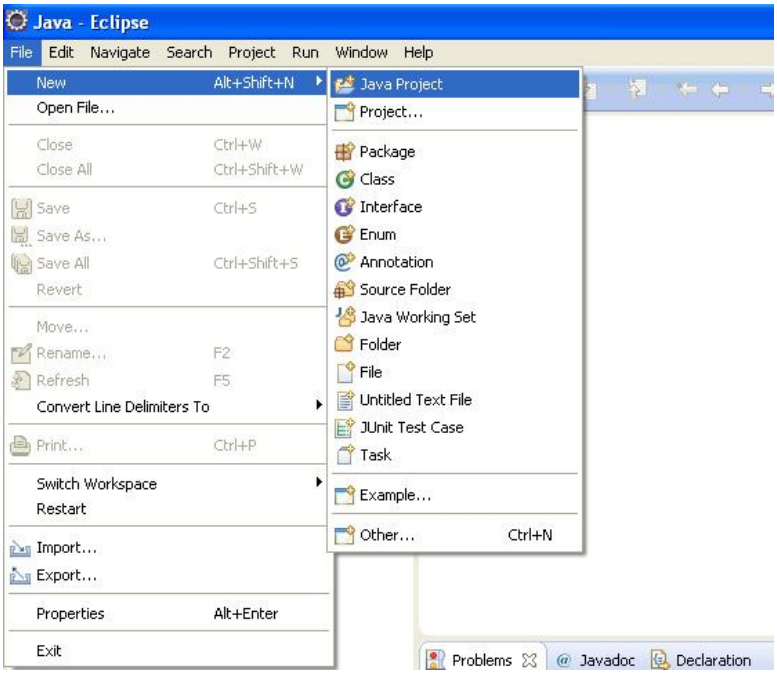
В Eclipse (и в другите подобни среди) има няколко основни панела (фиг. 1.4):

- В **„Управление на проекти"** се показват в отделни таб-ове „Package Explorer" и „Project Explorer", които представят различни изгледи към създадените проекти (файлова организация и организация на класове) и използваните в тях ресурси. **Включването на „Project Explorer" и други изгледи се извършва от менюто Window/Show View/Project Explorer.**
- В „Редактор" се редактира кода, като в различни таб-ове може да са отворени няколко файла едновременно;
- „**Дърво на текущия клас**" – представя основните членове на класа (полета и методи) и предоставя нагледен начин за по-лесен достъп до тях (при избор на елемента).
- Панела за „**резултат**" показва резултата в конзолата, но в отделни таб-ове могат да се видят проблеми, генерирана документация, декларации и др.



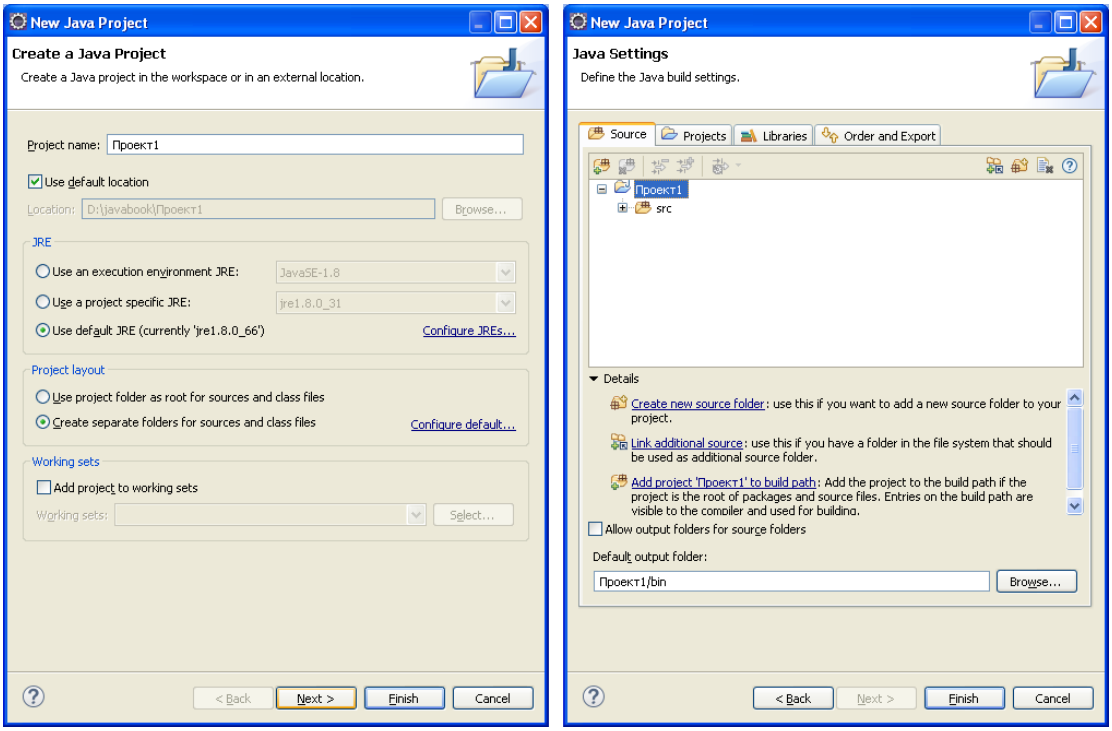
Фиг. 1.4. Основни панели в Eclipse (при Java SE перспектива)

За да напишем програма, трябва първо да създадем нов проект. Това става от меню File/New/Java Project.



Фиг. 1.5. Стартиране на помощник за създаване на нов проект

Единственото задължително нещо при създаване на проект е името му, което се задава в полето „Project name” (фиг. 1.6), след което може да се натисне бутона „Finish”. При това, в стандартното работно пространство (изписано в поле *Location*), ще се създаде директория с име, като името на проекта. В нея ще се записват всички файлове на проекта. В таб “Libraries” са указани включените в проекта стандартни архивирани (с jar) библиотеки на JRE. Има възможност да добавяме и допълнителни библиотеки както по време на създаване на проекта, така и на по-късен етап.

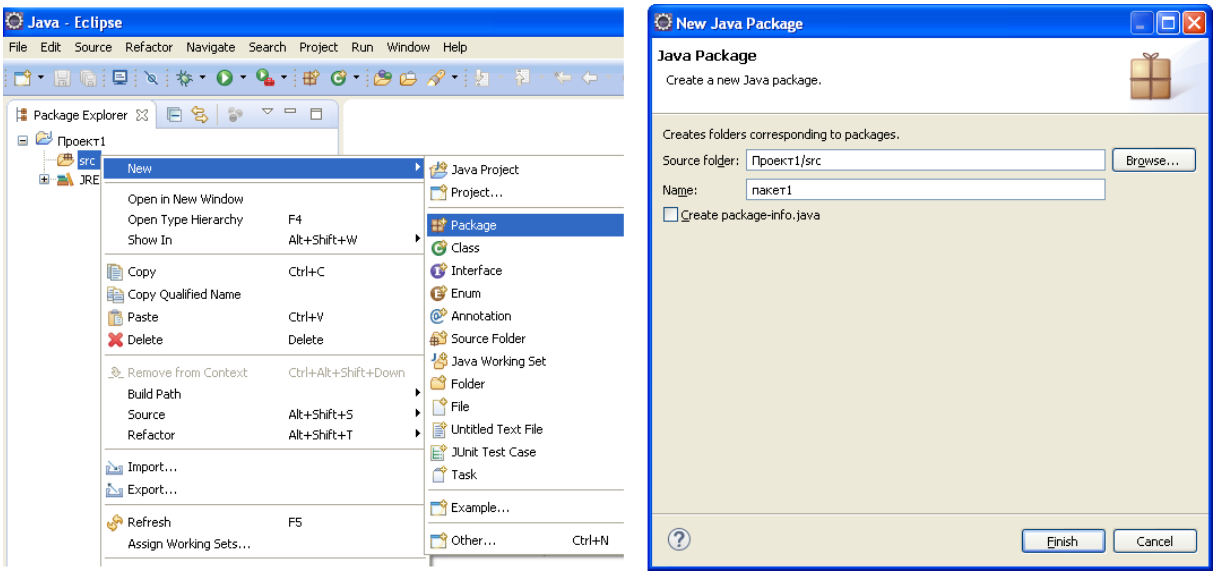


Фиг. 1.6. Създаване на нов проект

След като проектът е създаден, в него може да се добавят йерархия от пакети, при което се създават съответни директории. Във всяка директория/пакет може да има един или повече класове, както и други ресурси.

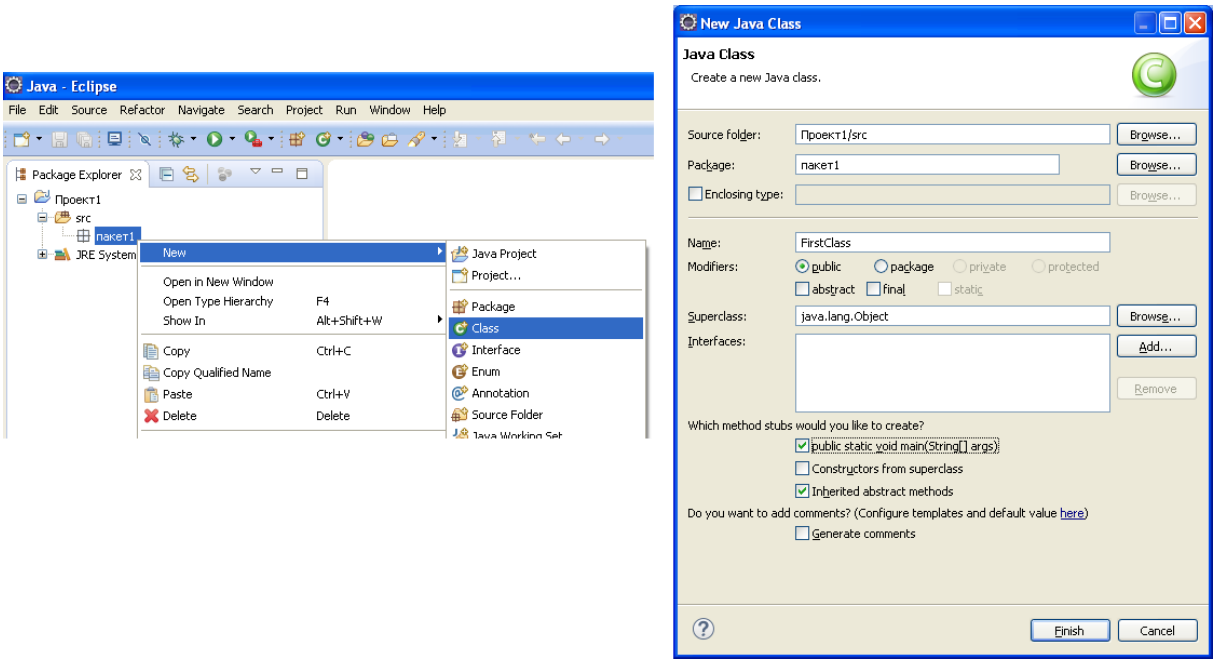
Нови елементи може да се добавят към проекта от менюто File или от контекстното меню, което е се визуализира след щракане с десния бутон на мишката върху проекта.

Нов пакет се добавя от меню New/Package, след което се задава името на пакета. На Фиг. 1.7. е показано създаването на пакет с име “пакет1”.



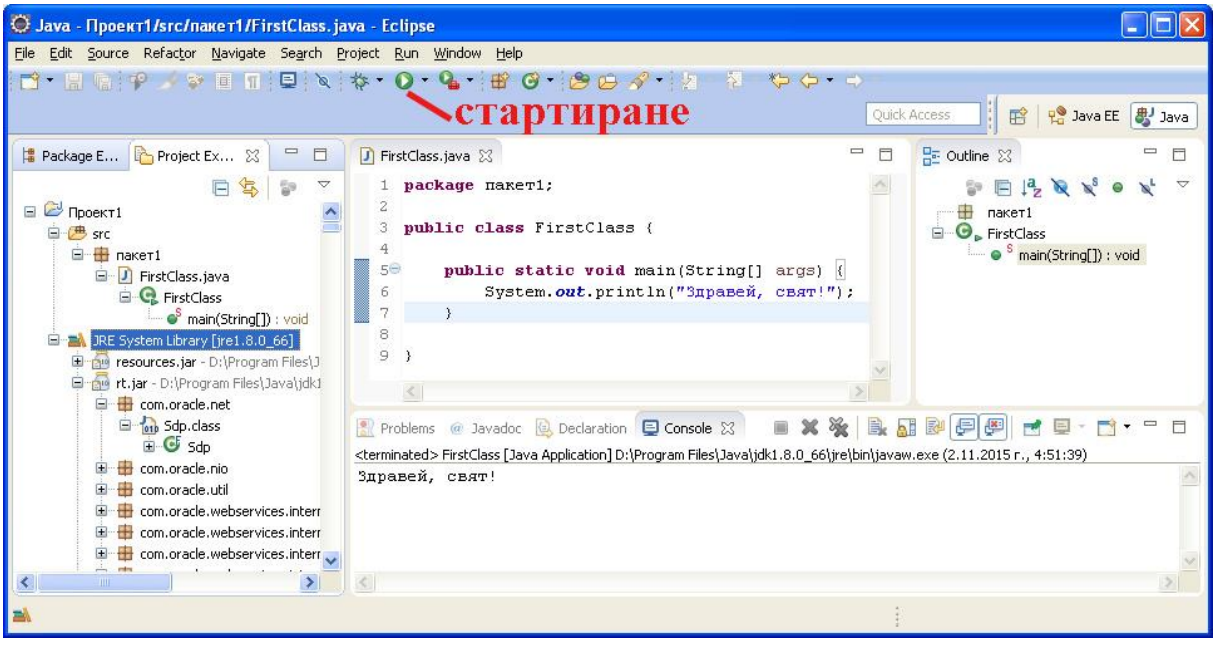
Фиг. 1.7. Създаване на нов пакет

По аналогичен начин (фиг. 1.8), в „пакет1” се създава нов клас с име FirstClass. Избира се New/Class, визуализира се прозорец, в който се задава име на класа, и ако желаем да се генерира автоматично main-метод, поставяме отметка. Автоматично се появява името на пакета (ако сме щракнали върху него с десния бутон на мишката) и по подразбиране е избрана опцията класа да е общодостъпен (полета Modifiers – “public”). Може да се задават и други характеристики на класа, които засега не е необходимо да променяме и поради това оставяме без подробни обяснения.



Фиг. 1.8. Създаване на нов клас

След натискане на бутон Finish, в редактора автоматично се генерира код за класа и main-метода, като остава да се попълни кода в тялото на main-метода (фиг. 1.9).



Фиг. 1.9. Стартиране на програма в Eclipse

Това, че един клас принадлежи към определен пакет, трябва да се укаже в кода на файла, преди описанието на класа. Извършва се (фиг. 1.9) с ключовата дума package последвана от името на пакета и „точка и запетая” (;). Ако сме указали пакет в описанието на java-файл, трябва да е създадена и съответна директория (което се прави автоматично от IDE-тата).

На фиг. 1.9 в таб „Console” се вижда резултата от изпълнението на програмата. Компилирането и последващото изпълнение на програмата се извършва от менюто Run/Run или с натискане на клавишната комбинация Ctrl+F11, или с бутона за стартиране, показан на екрана от фиг. 1.9. В изгледа „Project Explorer” е показано разгънато дървото на проекта, в което на различни нива от йерархията се виждат пакет, файл, клас, метод. Разгърната е и една част от дървото на библиотеките, в които има множество пакети, съдържащи компилирани класове (class-файлове). В библиотеката rt.jar се намират основната част от стандартните класове на Java. В изгледа “Outline” се показва дървото с елементи на текущия клас.

Променливи, типове, оператори, изрази

В езиците за програмиране, стойностите се описват с различни видове **величини**, най-често използваните от които са **променливи**. Всяка променлива си има **име, тип и стойност**. Типовете определят множество от възможни стойности. Съответно **променливите могат да приемат само измежду стойностите на указания им тип**.

В Java има примитивни и сложни типове. Примитивните типове са за числа, символи и логически (булеви) стойности:

- цели числа – byte, short int, long – които се различават по обхвата от числови стойности;
- реални числа с плаваща запетая – float, double;
- символ – char;
- логическа стойност – boolean.

Сложните типове са класове, интерфейси, изброими (enum) типове и др., като това включва и дефинирани от потребителите типове.

Класът String е специален клас за работа с низове (текст). С цел лесното му използване, за него са направени някои изключения, които го различават донякъде от обикновените класове на Java.

Променлива в Java се декларира, като се зададе **„тип” последван от „име”** и символа за край на оператор (;) напр.**:**

```
byte age;
double x;
```

С оператор равно (=) може да се присвоява стойност на променлива при нейното деклариране или в следваща част от кода:

```
byte age = 1; // променлива от тип byte с име age и стойност 1
double x;     // променлива от тип double с име x
x = 5.3;      // присвояване на стойност на x
String name = "Иван";
// Променлива за низ с име name. Низ е текст, заграден в кавички
```

Върху величини от конкретен тип могат да се прилагат различни видове оператори (аритметични, логически, за сравнение и др.), при което се създават изрази.

Например, аритметичните оператори +, - , * и / се използват при работа с числови величини. Оператор + може да се използва и за конкатениране (залепване) на низове, ако се използва с операнди низове. В следващата програма е показана работа с изрази:

- x= x+2 – стойността, получена от изчислението на израза x+2 се присвоява като нова стойност на x;
- низа, който отпечатваме, се образува като слепим стойностите на няколко величини – `name + " е на " + age + " г. и тежи " + x + " кг."`, т.е. към текущата стойност на променливата name залепваме последователно: низа „ е на ”; стойността на променливата age; низа „ г. и тежи ”; стойността на променливата x.

```
public class TestVariables {
      public static void main(String[] args) {
            byte age = 1;
            double x;
            x = 7.3;
            String name = "Иван";
            x = x + 2;
            System.out.println(name + " е на " + age + " г. и тежи " + x +
" кг.");
      }
   }
```

Резултат:

```
Иван е на 1 г. и тежи 9.3 кг.
```

Въведение в обектно-ориентираното програмиране с Java

Основни понятия в обектно-ориентираното програмиране (ООП) са класовете и обектите. **Обектите описват състоянието и поведението на същности (конкретни екземпляри) от реалния свят. Класът задава шаблон за създаване на обекти. Например „Човек” е клас, а „Иван” е обект от класа „Човек”.**

В реалния свят:

- ***състоянието (state)*** на обектите се описва чрез ***физически характеристики,*** наричани още свойства (properties);
- ***поведението (behavior)*** на обектите се описва чрез ***функционални характеристики*** – това са действия, които обектът може да изпълнява и които могат да имат различен резултат, в зависимост от конкретното състояние на обекта.

Например, човек може да се опише с:

- физическите характеристики [име, възраст в години]; възможни действия (поведение) са [кажи си името и възрастта, увеличи възрастта с 1].

Конкретен човек, може да се опише чрез конкретни стойности на физическите характеристики, напр. [Иван, 18].

Действията са свързани с избраните за описване характеристики. Например, за представения човек не може да има действие [кажи си датата на раждане], защото сред характеристиките за описание на човек няма „дата на раждане”.

В езиците за програмиране:

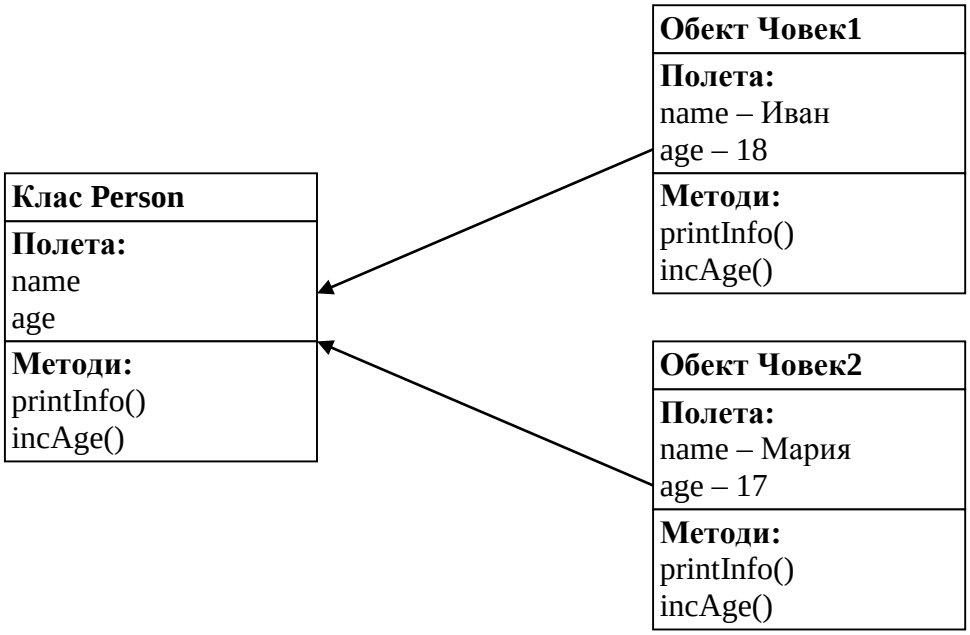
- ***състоянието*** се описва с **полета** от определени типове. В зависимост от езиците за програмиране и контекста, полетата може да се наричат и член-данни, атрибути, променливи на класа и др.;
- **поведението** се описва чрез **методи**, наричани още функции, процедури, член-методи, действия, операции.

Методите на един клас в общия случай извършват операции върху полетата на класа. Може да отпечатват резултата от работата си в някакъв изход – конзола, файл, база данни, или подобно на полетата, да изчисляват стойност от някакъв тип, която да се използва в последствие.

В термините на ООП:

- класовете описват полета и методи;
- обектите като представители на класовете, притежават всички полета и методи на класа. Те имат конкретни стойности за полетата и методите им обработват конкретните стойности на полетата.

На фиг. 1.10 илюстративно е представен класът за човек – Клас Person и два обекта – за Човек 1 и Човек 2.



фиг. 1.10. Клас Person и обекти от класа

Клас Person няма main метод, но може да се добави.

Една програма в общия случай се състои от няколко класа. Само един от класовете има main метод. При стартиране на програма, се изпълнява main метода, след което той може да предаде контрола на изпълнението на програмата на други класове.

```
public class Person {
	String name;
	int age;

	void printInfo(){
		System.out.println(name + " е на " + age + " г.");
	}
	void incAge(){
		age = age + 1;
	}
}
```

В методите на класа може да се работи с полетата на класа така, както се работи с обикновени променливи, дефинирани в метод.

Методите на класа Person не са статични (не са зададени с модификатор static), което означава, че са динамични. За разлика от статичните методи (какъвто е методът main), към динамични методи може да се обръщаме единствено чрез обект на класа.

Обекти на класа Person се декларират подобно на разгледаните по-горе декларации на променливи (с тип, последван от име).

Задаването на стойност за обект става като се използва оператор new, последван от извикването на **специален метод на класа**, наречен **конструктор**. Конструкторите на класа може да са няколко: всички имат име като името на класа, но трябва да имат различни по брой и/или тип параметри. Всеки клас си има един конструктор по подразбиране, който е без параметри. Ако в даден клас не е деклариран конструктор, то за създаване на обекти може да се ползва конструктора по подразбиране. Напр. следният код създава променлива с име p1, която е обект от клас Person.

```
Person p1 = new Person();
```

Достъпът до полетата и методите на един обект се осъществява посредством името на обекта и оператор „точка" (.). Така могат да се четат и обработват стойностите на полетата на обекта.

В следващия пример е дефиниран клас TestPerson и е показано как се работи с обекти от клас Person.

```
public class TestPerson {
      public static void main(String[] args) {
            Person p1 = new Person(); // работа с човек 1
            p1.name = "Иван";         // задаваме стойност за името...
            p1.age = 18;              // ... и възрастта на човек p1
            p1.incAge();              // извикват се методи,...
            p1.printInfo();           // ...свързани само с p1

            Person p2 = new Person(); // работа с човек 2
            p2.printInfo();           // извикване на метод за p2
      }
   }
```

Резултат:

```
Иван е на 19 г.
null е на 0 г.
```

След като „сме забравили" да зададем стойности на полетата на обект p2, в резултата се визуализират стойностите по подразбиране – тези, които са зададени от конструктора по подразбиране.

Основни принципи на ООП

Основните принципи на ООП са капсулиране, наследяване, абстракция и полиморфизъм.

Чрез наследяването се дава възможност за създаване на класове– наследници, които наследяват други, вече съществуващи класове. **Класовете наследници притежават всички характеристики на родителските класове, като имат и собствени полета и методи. Капсулирането позволява да се задават свойства за видимост (достъпност) на елементите на клас. Това определя дали** полетата и методите на даден клас могат да бъдат извиквани чрез обекти на класа, както и от класове от текущия или други пакети. Чрез **абстракцията** може да **задължим наследник да създаде метод с конкретно име,**

както и да дефинираме методи с еднакви заглавни части (декларации) в класовете от една йерархия на наследявания. Полиморфизмът дава възможност да се представя обект от един клас като обект от друг клас, намиращ се в същата йерархия от класове. Обикновено, полиморфизмът се използва за представяне на обекти на множество класове наследници като обекти на родителски клас. Мощен механизъм, който осигуряват наследяването, абстракцията и полиморфизма, е чрез обект от родителски клас да се обръщаме към методи декларирани в родителския клас, но дефинирани в наследник.

Чрез принципите на ОПП се осигурява възможността всички класове от една йерархия на наследяването да имат общо поведение (методи), декларирано от родителския клас, което да варира за различните класове.

Клас java.lang.Object

Класът java.lang.Object стои в основата на йерархията от наследявания на класове в Java. Дори наследяването да не е указано явно, всички класове в Java са преки или непреки негови наследници – имат поведение, повлияно от този клас и могат да бъдат управлявани еднотипно от ВМ на Java.

Java SE API

Съществуват различни API-та на Java, организирани в йерархии от пакети, класове и интерфейси. Всяко API предоставя функционалност в определена предметна област, която може да бъде ползвана както от потребителите, така и от API-та на по-горно ниво. Основните API-та на Java SE предоставят функционалности за вход и изход; математически константи и функции; колекции; интернационализация, осигуряваща механизми за многоезичност; дата и време; сериализация; XML; бази данни; отдалечено извикване на методи; създаване на приложения с графичен интерфейс; и много други по-специализирани дейности.

Документация на Java SE API

На адрес http://docs.oracle.com/javase/8/docs/api/ може да бъде разгледана документацията на осма версия на Java SE API. Тя съдържа описание на всички пакети, класове и интерфейси, полета и методи.

Коментарите са важна част от процеса на създаване на професионални приложения. Java поддържа специален вид коментари, наречени Javadoc, започващи с низа /** и завършващи с низа */. Те могат да съдържат HTML код и специални тагове (като @author, @version, @param, @return и др.) за описание на различни елементи от кода. Тези коментари се използват от инструмента Javadoc за автоматизирано генериране на документация в HTML формат. При това се създават външни препратки (връзки) към страници, описващи свързаните пакети и класове, както и вътрешни препратки към елементите на документирания клас.

Използване на клас в друг клас

За да може в един клас да бъде използван друг клас, вторият клас трябва изрично да бъде описан чрез ключовата дума import, последвана от пълното име на класа, включващо и пакета, в който се намира (например `import java.util.Random;`)

В един клас може да има няколко import-включвания. Със * се означават всички класове от даден пакет (например `import java.util.*;`). Включванията на класовете трябва да се записват в кода на програмата след декларацията на пакета и преди декларацията на класа.

```
package chapter1;

import java.util.Random;

public class Test...
```

Пакетът java.lang съдържа основни класове на Java, а в java.util има множество помощни класове.

Ако един клас използва друг клас от същия пакет, в първия клас не е необходимо да се указва включването на втория. Също така класове от java.lang не е нужно да се включват с import, защото се „виждат" автоматично.

Някои по-важни класове от java.lang са:

- **Object** – основен родителски клас за всички класове, предоставящ базова функционалност и възможност за еднотипно управление от ВМ на Java;
- **System** – съдържа системни методи за работа с потоци за вход (in), изход (out), грешка (err), системно време и др.
- **String** – клас за работа с низове, предоставящ различни методи за обработка като търсене на подниз, замяна на част от низ с друг низ, преобразуване в малки или главни букви и много други;
- **Math** – предоставя методи за работа с математически константи и функции.

Въпроси и задачи за упражнения

1. Инсталирайте подходяща за вашия компютър дистрибуция на Java Platform SE. При необходимост използвайте съответната документация: http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Разгледайте изпълнимите файлове в bin директориите на създадените за JDK и JRE папки и потърсете информация за използването им.
3. Инсталирайте Eclipse.
4. Създайте проект с йерархия от пакети на три нива. В пакет от трето ниво създайте описаните в главата класове. Разгледайте създадените директории и файлове.
5. Разгледайте менютата на Eclipse. Включете Project Explorer и други изгледи.
6. Разгледайте в Project Explorer файловете на включените в проекта системни библиотеки.
7. Разгледайте API-тата на Java SE.
8. Потърсете в Интернет и разгледайте техники за писане на коментари за Javadoc Tool.
9. Разгледайте и тествайте методите на класа java.lang.Math.

Глава 2. Основи на програмирането

Програмирането е част от една по обширна област – **компютърната информатика**, която изследва начини за **автоматизирано събиране, обработване, съхранение и разпространение** на **информацията**.

Информацията е **знание, сведение, факт за обект, процес, модел**... В компютрите **информацията се представя чрез цифрови данни**. Основна **единица за количественото измерване на информацията е бит** (bit – идва от binary digit). Един бит може да приема стойности 0 и 1. Цифрите 0/1 могат да се интерпретират по различни начини: не/да; има напрежение/няма напрежение; истина/лъжа. За по-лесна обработка, битовете се групират в групи от по осем бита, наречени байтове и са дефинирани техни производни мерни единици:

бит	**байт**
1 килобит – 1024 бита – 2^{10} бита	1 килобайт – 1024 байта
1 мегабит – 1024 килобита – 2^{20} бита	1 мегабайт – 1024 килобайта
1 гигабит – 1024 мегабита – 2^{30} бита	1 гигабайт – 1024 мегабайта
1 терабит – 1024 гигабита – 2^{40} бита	1 терабайт – 1024 гигабайта

Математически основи на програмирането

Извършването на всички автоматизирани действия, свързани с информацията, се базират на строги математически основи – бройни системи, логика, булева алгебра и се реализират чрез различни алгоритми. Математиката се използва за реализацията на основни „елементарни" действия като:

- представяне на цели и реални числа в компютърните системи и извършване на аритметични и логически действия с тях;
- преобразуване на числа от една в друга бройни системи, като най-използваните са двоична, десетична, осмична, шестнадесетична;
- представяне цветовете на пикселите на екрана чрез числа в различни системи за кодиране – RGB (Red Green Blue – червен зелен син), RGBA (Alpha – прозрачност), CMYK (Cyan Magenta Yellow blacK - Циан Пурпурен Жълт Черен).
- представяне на буквите и символите на естествените езици в различни системи за кодиране (ASCII, UNICODE и много други) и изобразяването им на екрана.

При решаването на конкретни практически задачи се използват много други области на математиката: алгебра, геометрия, теория на графите, теория на вероятностите, статистика, оптимиране...

Алгоритми, програми, приложения

В процеса на развитие на хардуерните и софтуерните възможности на компютърните системи, са реализирани множество интерфейси (функции, API-та) на различни нива – хардуерно, операционна система, език за програмиране. Всяко от тези нива скрива работата на по-ниското ниво и предоставя механизми за достъп на по-високите нива. Всяко следващо ниво на стандартните интерфейси предоставя на програмистите по-лесен начин за създаване на сложни **програми и приложения**.

Програмите обикновено се състоят от един модул и имат една входна точка за начало на изпълнението си. **Приложенията може да се състоят от много модули**, които работят заедно; може да имат различни входни точки и различните модули може да са написани на различни езици. Всички програмите са приложения, но не всички приложения са програми.

Логиката на програмите и приложенията се описва чрез алгоритми.

Алгоритъм е описание на последователност от действия, които решават определена задача или клас от задачи. Алгоритмите може да се представят графично (като изображения) или да бъдат описани на естествен език. За да бъдат разбираеми за компютърна система, трябва да бъдат написани на **машинен език** за определен вид процесори.

Езици за програмиране

В програмирането, алгоритмите се описват на конкретни езици за програмиране. Те се състоят от действия, наречени команди, операции или инструкции.

Езиците за програмиране са **изкуствени езици**, които се развиват заедно развитието на компютърните системи. Може да се категоризират на няколко **нива:**

- ***Машинни езици*** (първо ниво) – програмите се описват като последователност от нули и единици, съдържащи директни команди (инструкции) за конкретен тип процесори;
- ***Асемблерни езици*** (второ ниво) – имената на инструкциите се представят чрез символни кодове; данните могат да се представят като текст, десетични или шестнадесетични числа;
- ***Процедурно-ориентирани езици*** (трето ниво) – кодът за създаване на програмите съдържа инструкции, които се доближават до естествен език. Алгоритмите се описват чрез процедури, които могат да се извикват и изпълняват многократно. Процедурите работят само с параметри и глобални данни.
- ***Обектно-ориентирани езици за програмиране*** (четвърто ниво) – използват съвсем различна концепция за създаване и структуриране на програми. Всеки обект (понятие, процес, модел...) от реалния свят се описва чрез множество от физически и функционални характеристики (полета и методи). Обектите комуникират помежду си чрез изпращане на съобщения (извиквания на методи).

Езиците от трето и четвърто ниво се наричат езици от високо ниво. Към трето ниво спадат и проблемно-ориентираните езици като Пролог и Лисп, при които програмите се описват като множество от факти и правила. Обектно-ориентираният подход дава възможности за развитие на мощни визуални средства за създаване на приложения, с малко писане на код.

Транслатори

Компютрите могат да изпълняват само програми, написани на машинен код. Поради тази причина е необходимо програмите, написани на език от високо ниво, да бъдат преобразувани до машинен код.

Това става с помощта на транслатори (translator – преводач). Те са два вида:

- ***Компилатор*** – създава изпълним файл, който може да бъде изпълнен самостоятелно по всяко време. Ако в изходния код на програмата има синтактични грешки, компилаторът не може да създаде изпълним файл.
- ***Интерпретатор*** – интерпретира (тълкува) изходния (сорс) код, без да създава изпълним файл.

Интерпретирането на изходен код и преобразуването му в машинни инструкции е по-бавен процес, в сравнение с изпълнението на предварително компилирана програмата.

Компилатор и ВМ на Java

Java комбинира двете техники за транслация. Компилаторът на Java компилира изходния код до междинен код, наречен байткод, който съдържа инструкции не за конкретен реален процесор, а за Виртуалната Машина (ВМ) на Java.

Изходният код на Java се съхранява във файлове с разширение ".java". При липса на синтактични грешки в изходния код, компилаторът на Java създава файлове с разширение ".class", съдържащи байткод. Интерпретаторът на Java, наречен ВМ на Java, изпълнява байткод.

Изпълнение на програмите – статична и динамична памет

Изпълнимите програми съдържат множество от инструкции, описани в т. нар. стек на програмата, който съдържа променливи, действия върху тях, подпрограми и др.

Паметта, необходима за статичните променливи, се заделя предварително в стека. Тя се използва дори и при промяна на стойностите им.

Като стойности на динамичните променливи – указатели и референции – се записват адреси на динамични обекти, които се създават по време на изпълнение на програмата (runtime). За самите динамични обекти се заделя памет в т. нар. динамична памет (хийп).

Обикновено и статичната, и динамичната памет на една програма са обособени части в RAM паметта.

Тъй като програмите в Java се зареждат динамично във ВМ на Java, то статичната и динамичната им памет са части от динамичната памет на ВМ.

Статични и динамични променливи в Java

За всички елементарни типове в Java се създават статични променливи, а за сложните обекти се заделя динамична памет. Името на променливата на динамичен обект е само референция. Това означава, че една променлива може да „сочи" (реферира, да се свързва с) различни динамични обекти по време на изпълнение на програмата.

Тъй като при създаването на динамични обекти може да се натрупат множество излишни динамични обекти, които не са свързани с променлива, то възниква въпросът за освобождаване на заетата динамична памет.

Освобождаване на заетата памет. Java Garbage Collector

Освобождаването на статичната памет става автоматично, след приключване на изпълнението на програмата.

Освобождаването на паметта, заета от динамичен обект, става по различни начини в различните езици за програмиране.

Например, в C++ паметта се освобождава с помощта на оператор delete, за записването на което трябва да се грижи програмистът.

В Java, освобождаването става автоматично (без конструкции в езика за програмиране), с помощта на механизъм за почистване, наречен Garbage Collector (събирач на боклук). Той се пуска автоматично от ВМ на Java.

Компютърна памет

Най-малките клетки от паметта в компютрите са битовете. Един бит има две възможни стойности – 0 и 1. Осем бита образуват блок, наречен байт. Най-малкото двоично число, което може да се запише в един байт се състои от осем нули – 00000000, а най-голямото – от 8 единици – 11111111. Те съответстват на десетичните числа 0 и 255. За записване на по-големи числа се използват няколко последователни байта.

Логически се приема, че клетките в паметта се подреждат последователно. Всяка клетка си има адрес, което е цяло число, като адресирането започва от 0. В зависимост от адресите, с които могат да работят процесорите, има 16-битови, 32-битови, 64-битови и др. компютри. 16-битовите процесори могат да адресират RAM памет с размер до 64 килобайта (=2^{16} байта = 65536 байта), 32-битовите – до 4 гигабайта (= 2^{32} байта = 4 294 967 296 байта), 64-битовите – много повече. Операционните системи могат и налагат ограничения в обема на адресираната памет.

Бройни системи

Бройните системи са метод за представяне на числа, включващ графични знаци и правила за записване на числата. Една част от графичните знаци служат за означаване на цифри, а други (като десетична запетая) са спомагателни.

Бройните системи са два вида – позиционни и непозиционни. При позиционните, стойността на цифрата зависи от мястото ѝ в числото, за разлика от непозиционните.

Най-известна от непозиционните бройни системи е римската. Няма да се спираме подробно на тях, тъй като не са удобни за използване в компютърните системи, поради невъзможността чрез тях да се представи произволно число.

Арабската десетична бройна система, която използваме традиционно, е позиционна, и е с основа числото 10. Има множество други позиционни бройни системи, с различни основи . В компютърната техника се използват основно бройни системи с основа 2, 8, 10, 16 .

Стойността на една цифра в позиционна бройна системи зависи от позицията ѝ в записа на числото. Напр. в числото 123, първата цифра 1 се възприема като сто, 2 като двадесет, а 3 – като три.

Множителят, с който се изменя стойността на една цифра, се нарича **основа на бройната система.** Основата съвпада с броя използваните цифри.

Десетичната бройна система е с основа 10 и използва цифрите 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Цифрите в записа на едно число имат различна стойност, която е степен на основата 10. Напр. в записа на цяло число, най-дясната цифра представя единиците (10^0), следващата е за десетиците (10^1), после за стотиците (10^2) и т.н. Т. е. всеки разряд е 10 пъти по-голям от следващия го (в дясно) и 10 пъти по-малък от предшестващия го (от ляво). Напр.

$$123 = 1*10^2 + 2*10^1 + 3*10^0$$

В съвременните компютри се използва основно двоичната бройна система, тъй като в електрониката е лесно и евтино да се реализират логически схеми с две устойчиви състояния. Тя се състои от две цифри – 0 и 1 (означават съответно „няма ток", „има ток"), чрез които може да се представи всякаква информация. За извършване на различни действия се използва двоична аритметика. Числата се четат от ляво на дясно – от старшия към младшия разряд.

При едновременна работа с няколко бройни системи, за да не се допускат недоразумения, след всяко число, чрез долен индекс, се отбелязва бройната система. Напр., $84_{(10)}$, $1010_{(2)}$.

При двоичната бройна система важат същите правила, като за десетичната бройна система Тук обаче се използват само цифрите 0 и 1 и всяка съседна цифрова позиция от ляво надясно е с нарастваща степен на числото 2. Напр.:

$$\begin{aligned} 110101_{(2)} &= 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \\ &= 1*32 + 1*16 + 0*8 + 1*4 + 0*2 + 1*1 \end{aligned}$$

$= 32 + 16 + 4 + 1$
$= 53_{(10)}$

В двоичен код може да се записват не само цели числа, но и числа със знак и дробни числа.

За записване на числа със знак се използва специален бит за означаване на знака. Ако битът за знак има стойност 0, числото е положително, ако стойността му е 1 – числото е отрицателно. Така, ако за записа на цяло число се отделя 1 байт, в първия бит се записва знака, а в останалите 7 бита – самото число. Обхватът на целите числа със знак, които могат да се запишат в един байт е от $-2^7 + 1$ до 2^7, т.е. от -127 до 128.

Пример за положително число: $01001101_{(2)} = 77_{(10)}$,

Положителните числа се представят в прав код. Отрицателните числа се представят в обратен (допълнителен) код, при който всички битове се инвертират (заменя се 0 с 1 и 1 с 0) и към полученото число се добавя 1. Ползването на двете кодирания е с цел нулата да има само едно представяне.

За да получим $-77_{(10)}$, например, инвертираме побитово $77_{(10)}$ и добавяме 1:

$\sim 01001101_{(2)} + 00000001_{(2)} =$
$10110010_{(2)} + 00000001_{(2)} =$
$10110011_{(2)} = -77_{(10)}$

Със символът ‘~’ се означава оператора побитово отрицание.

За запис на дробни числа се използва позиционна точка, която съответства на десетичната точка в математиката. Цифрите вляво от позиционната точка означават цялата част на числото, а тези вдясно – дробната част. Цялата част на числото се изчислява по разгледания вече начин. Дробната част се изчислява, като цифрите се умножават с отрицателни степени на числото 2 т.е. първата цифра след позиционната точка се умножава с 2^{-1}, втората с 2^{-2}, третата с 2^{-3} и т.н. Например:

$1010.0110_{(2)} = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$
$+ 0*2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4}$
$=8 + 2 + 1/4 + 1/8 = 10 + 3/8 = 10\ 3/8$

Операции върху двоични числа

Аритметичните операции за събиране и изваждане на двоични числа се извършват по същия начин, както и при десетичните – поразредно. Операцията между i-тите разряди на двата операнда формира i-тия разряд на резултата. Понякога има пренос на единица към по-старшия разряд при събиране и заемане от по-старшия разряд при изваждане. Примери:

$1001_{(2)}$	$10100_{(2)}$
$+\ 1101_{(2)}$	$-\ 1011_{(2)}$
$= 01110_{(2)}$	$= 1001_{(2)}$

С двоичните числа може да се извършват и **побитови операции** - отрицание (**~**), "И" (**&**), "ИЛИ" (|) и изключващо "ИЛИ" (^). Правилата са дадени в следващата таблица.

a	b	NOT a (~a)	a AND b (a&b)	a OR b (a|b)	a XOR b (a^b)
0	**0**	1	0	0	0
0	**1**	1	0	1	1
1	**0**	0	0	1	1
1	**1**	0	1	1	0

Побитовото отрицание (NOT) се прилага върху един операнд и променя стойността на всеки бит на операнда. Битовете, които са имали стойност 0, се променят на 1, а тези със стойност 1, стават 0. Така се образува допълнението на даденото двоично число. Пример:

```
NOT 110100
  = 001011
```

Побитово „И” (AND) има стойност 1, само ако и двата операнда имат стойност 1, в противен случай стойността му е 0. Може да се използва да се провери стойността на даден бит. Напр. ако искаме да проверим каква е стойността на третия бит в числото 100110, може да извършим следната операция:

```
    100110
AND 000100
  = 000100
```

Тъй като резултатът не е 0, това ще означава че третият бит в нашето число е вдигнат, т.е. има стойност 1. Това се нарича **битово маскиране.** Операторът може да се използва да се „свалят” определени битове. За целта се използва втори операнд, който има стойност 0 във всички битове, които трябва да се свалят, а в останалите битове имат стойност 1.

Побитово „ИЛИ” (OR) връща 0 само ако и двата операнда едновременно имат стойност 0, в противен случай връща 1. Може да се използва за „вдигане” на определени битове.

Напр. ако искаме да вдигнем 2 и 5 бит на числото 1000100, може да извършим следната операция:

```
    1000100
OR  0010010
  = 1010110
```

Побитово „ИЗКЛЮЧВАЩО ИЛИ” (XOR) връща резултат 0, ако двата бита имат една и съща стойност, и 1 – ако са с различна стойност. Операторът може да се използва за сравняване на битове. Напр.

```
    0010
XOR 1000
  = 1010
```

Това означава, че 1-ви и 3-ти бит на двете числа са еднакви.

Този оператор също може да е използва за обръщане на битове. За целта трябва да се използва операнд, който има стойност 0 в тези позиции на битовете на целевото число, които трябва да се обърнат.

Специфични побитови оператори са побитово отместване вляво (<<) и побитово отместване вдясно (>>). Те извършват отместване наляво или надясно с указан брой позиции на всички битове на едно число. Тези оператори изискват два операнда – левият е числото, върху което ще се извърши операцията, а десния указва броя на отместване на битовете. Например: 1101001 << 3 означава отместване наляво с 3 бита, като резултата ще е 1101001000, а 1101001 >> 3 изисква отместване надясно с 3 бита, тук в резултат ще се получи 0001101.

Логически побитовото отместване вляво съответства на операцията умножение, а отместването вдясно – на деление. Това е поради факта, че в записа на едно число всеки ляв бит има стойност, 2 пъти по-висока от бита отдясно т.е. преместване на 1 бит наляво умножава числото по 2^1, преместване на 2 бита извършва умножение по 2^2, на 3 бита – 2^3 и т.н.

Недостатък на двоичната бройна система е дългият запис на цифрите. Често за представянето им, освен десетична се използват осмична и шестнадесетична бройни системи, поради факта, че 8 и 16 са степени на двойката (съответно 2^3 и 2^4).

Осмичната бройна система използва цифрите от 0 до 7. Напр.

$$\begin{aligned} 6021_{(8)} &= 6*8^3 + 0*8^2 + 2*8^1 + 1*8^0 \\ &= 6*512 + 0 + 16 + 1 \\ &= 3072 + 17 \\ &= 3089_{(10)} \end{aligned}$$

Шестнадесетичната бройна система използва 16 символа – цифрите от 0 до 9, A, B, C, D, E и F. Първите 6 букви от английската азбука означават числата от 10 до 15, както следва: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. Напр.

$$\begin{aligned} 6B2_{(16)} &= 6*16^2 + 11*16^1 + 2*16^0 \\ &= 6*256 + 176 + 2 \\ &= 1714_{(10)} \end{aligned}$$

Преобразуване на числа от една бройна система в друга

Всяко число може да се преобразува от една бройна система в друга. В компютърните системи преобразуванията се налагат, за да може компютърните двоични числа да се представят в удобния за човека десетичен вид и обратно – въведените от хората десетични числа да се запишат в компютъра като двоични.

Преобразуването на едно число от десетична в двоична бройна система, изисква деление на основата, в случая 2, като се записват последователно остатъците. Ако числото се дели на 2, се записва остатък 0, а ако не се дели – остатъкът е 1. След като деленето приключи, остатъците се записват в ред, обратен на реда, в който са получени и това е числото в двоична бройна система.

Напр. нека намерим как изглежда десетичното число 53 в двоична бройна система.

53 : 2 = 26, остатък 1,
26 : 2 = 13, остатък 0,
13 : 2 = 6, остатък 1,
6 : 2 = 3, остатък 0,
3 : 2 = 1, остатък 1,
1 : 2 = 0, остатък 1.

Така намираме, че $53_{(10)} = 110101_{(2)}$.

Ако е необходимо да се извърши преобразуване на число от една бройна система в друга, при което основата на едната бройна система е точна степен на другата, може да се използва друг алгоритъм. Нека разгледаме два примера за преобразуване на числа между двоична и осмична бройна система.

Ще ни е необходима таблица на съответствията между двете бройни системи:

Осмична	0	1	2	3	4	5	6	7
Двоична	000	001	010	011	100	101	110	111

Пример: Да представим двоичното число 11011101 в осмична бройна система.

Първо разделяме двоичното число на тройки цифри, като започнем отдясно на ляво. Ако най-лявата група се състои от по-малко от 3 цифри, дописваме необходимия брой нули отляво. За всяка тройка цифри търсим в таблицата съответната цифра в осмична бройна система:

$11011101_{(2)} = 11|011|101 = 011|011|101 = 335_{(8)}$

Пример: Ако желаем да запишем осмичното число 247 в двоична бройна система, заместваме всяка една от осмичните цифри със съответната тройка двоични цифри. Ако вляво на полученото двоично число има водещи нули, ги премахваме.

$247_{(8)} = 010|100|111 = 10100111_{(2)}$

Друг частен случай е, ако основите на двете бройни системи са точни степени на трето число. В този случай може да се използва помощна бройна система – с основа третото число. Например, осмичната и шестнайсетичната бройни системи са с основи, степени на числото 2 ($8 = 2^3$, $16 = 2^4$). Таблиците за съответствията между цифрите от двете бройни системи и помощната изглеждат така:

Шестна-десетична	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Двоична	000	001	010	011	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Осмична	0	1	2	3	4	5	6	7
Двоична	000	001	010	011	100	101	110	111

Преобразуването на число от едната бройна система в другата, изисква с помощта на таблиците, числото първо да се преобразува в помощната бройна система, и след това в целевата.

Пример: Да се запише осмичното числото 528 в шестнадесетична бройна система.

$$
\begin{aligned}
536_{(8)} &= 101|011|110_{(2)} \\
&= 101011110_{(2)} \\
&= 1|0101|1110_{(2)} \\
&= 0001|0101|1110_{(2)} \\
&= 15E_{(16)}
\end{aligned}
$$

Кодови таблици

Кодовите таблици на символите описват множество двойки от вида „число-символ”. Те дават връзката между символ и число, чрез което той се представя. Символите са букви от естествените езици, цифри, аритметични оператори, специални символи (интервал, tab, ins, del, край на ред...) и др.

Един и същи символ може да има различни кодове в различни кодови таблици.

При запис на конкретен символ, в паметта се записва неговия числов код (във вид на нули и единици) от конкретна кодова таблица.

За да бъде разчетена правилно дадена последователност от нули и единици, трябва да се знае каква кодовата таблица е използвана за кодирането ѝ. Чрез нея се определят групичките битове, сформиращи кода на един символ, а в зависимост от кода се изобразява съответния му символ.

Разбира се, текстообработващите програми скриват тези детайли от потребителите.

ASCII кодова таблица

Първата компютърна кодова таблица е ASCII (American Standard Code for Information Interchange). Тя е 7-битова и дефинира записа на 128 стандартни символа, базирани на латинската азбука. Разширената 8 битовата версия (Extended ASCII) позволява записването на специфични езикови символи на избран естествен език. При това обаче, един и същи код може да има различни съответни символи за различните езици. Следващата схема показва ASCII кодовата таблица (източник: http://www.cpptutor.com/ascii.htm).

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

UNICODE кодова таблица

На базата на ASCII кодовата таблица са създадени множество други кодови таблици (описани в стандарти ISO-8859-n, Windows-n и др.), включващи освен основните 128 символа и символи от специфични езици.

Поради необходимостта от стандартизиране, в последствие е създадена универсалната кодова таблица UNICODE. Тя е многоезична и описва символите на всички естествени езици, както и всякакви специални символи.

UNICODE е 16 битова и чрез нея може да се записват 65536 (2^{16}) броя символа. Java използва UNCODE. В следващата таблица е показана част от кодовата таблица UNICODE, отнасяща се за българските букви.

1040 - А	1056 - Р	1072 - а	1088 - р
1041 - Б	1057 - С	1073 - б	1089 - с
1042 - В	1058 - Т	1074 - в	1090 - т
1043 - Г	1059 - У	1075 - г	1091 - у
1044 - Д	1060 - Ф	1076 - д	1092 - ф
1045 - Е	1061 - Х	1077 - е	1093 - х
1046 - Ж	1062 - Ц	1078 - ж	1094 - ц
1047 - З	1063 - Ч	1079 - з	1095 - ч
1048 - И	1064 - Ш	1080 - и	1096 - ш
1049 - Й	1065 - Щ	1081 - й	1097 - щ
1050 - К	1066 - Ъ	1082 - к	1098 - ъ
1051 - Л	1067 - Ы	1083 - л	1099 - ы
1052 - М	1068 - Ь	1084 - м	1100 - ь
1053 - Н	1069 - Э	1085 - н	1101 - э
1054 - О	1070 - Ю	1086 - о	1102 - ю
1055 - П	1071 - Я	1087 - п	1103 - я

Шрифтове

Шрифтовете определят графични представяния на символите от кодовите таблици. Един символ може да има различни графични представяния за различните шрифтове.

Шрифтовете имат характеристики, чрез които може да се променя изображението им: размер на буквите, тегло (normal, bold и др.), наклон (normal, italic и др.) и др. Не е задължително един шрифт да има графично представяне за всички символи, които могат да се запишат чрез използваната кодова таблица.

Някои по-известни шрифтове са Times New Roman, Verdana, Arial, Courier...

Булева алгебра

Булевата алгебра възниква през 1847 г., когато Джордж Бул, английски учител по математика, се опитал да формализира логиката с цел въвеждане и използване на алгебрични методи в логиката.

Основен обект на изследване в булевата алгебра са **логическите съждения**. Те са твърдения, които могат да се оценят като – истина (true, 1) или лъжа (false, 0). Напр. „България е държава” е съждение, което има стойност истина, а „България е континент” е съждение, със стойност лъжа. Изреченията „Къде се намира България?” и „Само ако можех да спечеля 1 милион лева от тотото!” не са съждения.

Има изречения, които в определен момент не могат да бъдат оценени като истина или лъжа. Такива твърдения се наричат **предикати**. Напр. „Числото x е четно”. Ние не знаем кое е числото x, затова не можем да определим дали е четно или нечетно. При зададена конкретна стойност на променливата x, напр. x = 20, изречението „Числото 20 е четно” става съждение, което може да се оцени като вярно.

Съжденията биват прости и съставни. **Просто съждение** е съждение, което се формулира чрез просто изречение и утвърждава или отрича един признак. **Съставното съждение** се състои от две или повече прости съждения. Конструира се с помощта на логически съюзи и словосъчетания – „не", „и", „или", „ако ... то", „тогава и само тогава, когато" и т.н. Напр. „Земята е кръгла и Луната се върти около Земята" е съставно съждение.

В булевата алгебра има две **логически константи** – 0 и 1. **Логическа променлива** е величина, която може да приема различни стойности във времето – 0 или 1. **Логическа (булева) функция** е функция, чиято стойност зависи от краен брой логически променливи. Понякога логическите функции се наричат логически оператори. Логиката на по-известните от тях е подобна (но не идентична) на представената логика на побитовите оператори, и в някои случаи се означават с подобни символи.

Множеството от конкретни стойности на променливите на дадена логическа функция се нарича **набор**. Една логическа функция може да се дефинира по няколко начина - **словесно** (чрез текст), **таблично** (чрез таблици на истинност), **аналитично** (чрез формула) или **графично** (чрез графика).

Табличното представяне на булева функция изисква явно задаване на всевъзможните набори от допустими стойности на променливите, и съответната стойност на функцията.

Ще разгледаме накратко някои булеви функции на една и две променливи.

Функции на една булева променлива

Броят на възможните набори на функция на една променлива са два, а броят на функциите – четири. Те са представени в следващата таблица.

p	$F_1(p)$ Идентитет	$F_2(p)$ Отрицание	$F_3(p)$ Противоречие	$F_4(p)$ Тавтология
0	0	1	0	1
1	1	0	0	1

Идентитетът е функция, която повтаря стойностите на променливата величина. При **отрицанието**, функцията приема стойности, противоположни на стойностите на променливата. Функция, която има стойност 0 („невярно"), без значение от конкретната стойност на променливата, се нарича **противоречие**, а ако стойността ѝ е винаги 1 („вярно"), функцията се нарича **тавтология**.

Функции на две булеви променливи

Таблицата на истинност на функция на две булеви променливи включва четири набора на променливите и шестнадесет на брой функции. По-важните функции са представени в следната таблица.

p	q	конюнкция (и) $p \wedge q$	дизюнкция (или) $p \vee q$	импликация (следствие) $p \rightarrow q$	еквивалентност $p \leftrightarrow q$	сума по модул 2 (изключващо или) $p \oplus q$	стрелка на Пирс (логическо или-не) $p \downarrow q$	щрих на Шефер (логическо и-не) $p \mid q$
0	**0**	0	0	1	1	0	1	1
0	**1**	0	1	1	0	1	0	1
1	**0**	0	1	0	0	1	0	1
1	**1**	1	1	1	1	0	0	0

Логиката на функциите е следната:

- **Конюнкция (логическо И)** – резултатът е истина, само ако и двата операнда (p и q) са истина; в противен случай е лъжа.
- **Дизюнкция (логическо ИЛИ)** – резултатът е истина, ако поне единия от двата операнда (p или q) е истина; в противен случай е лъжа.
- **Импликация (следствие)** – има стойност истина в два случая – ако от „лъжа следва лъжа или истина" и „от истина следва истина"; лъжа е само случая, че „от истина следва лъжа".
- **Еквивалентност** – резултатът е истина, ако двата операнда (p и q) имат една и съща стойност; в противен случай е лъжа.
- **Сума по модул 2 (изключващо или)** – резултатът е истина, ако двата операнда (p и q) са различни; иначе е лъжа. Друга логика (в контекста на „изключващо или") е: резултатът е истина, ако само първия или само втория операнд са истина; в противен случай е лъжа. Трети вариант за интерпретация, свързан със „сума по модул 2" е, че функцията е подобна на побитово сумиране – 0+0=0; 0+1=1; 1+0=1; 1+1=(1)0. В резултата на последното равенство остава само нулата, а единицата преминава в по-старшия разряд.
- **Стрелка на Пирс (логическо ИЛИ-НЕ)** – отрицание на дизюнкцията.
- **Щрих на Шефер (логическо и-не)** – отрицание на конюнкцията.

Някои логически функции могат да се изразят чрез други логически функции. **Съвкупност от краен брой логически функции, чрез които могат да се представят всички останали логически функции, се нарича функционално пълна система от логически функции.**

Функционално пълни системи са например комбинациите от:

- отрицание и конюнкция;
- отрицание и дизюнкция;
- и др.

Най-голямо приложение в компютрите и езиците за програмиране намира т. нар. **класически базис**, който се състои от функциите **конюнкция, дизюнкция и отрицание**. По-рядко използван, но зададен като възможен за използване оператор в езиците за програмиране е **„Изключващо ИЛИ"**. Тези четири оператора (функции) не са минимален базис, но са естествени и удобни за използване от хората.

Има различни математически означения за посочените четири оператора. В следващата таблица са показани някои често използвани означения, използвани в различни езици за програмиране.

Функция/ езици	**Отрицание**	**Логическо И**	**Логическо ИЛИ**	**Изключващо ИЛИ**
В повечето ЕП, Java	!	&&	||	^
В някои ЕП	NOT	AND	OR	XOR

Съждителни смятане

Съждителен израз е съвкупност от съждителни променливи a, b, c и т.н., свързани със знаци за логически операции (!, &&, || и др.) и скоби, за указване приоритета на операциите. От съществено значение е да се прави разлика между съждение и съждителен израз. Съждителният израз описва цял клас от съждения със сходна структура. Напр. съждителният израз p||q изразява логическата структура на следните съждения:

„Едно изречение може да бъде просто или съставно"

„Утре времето ще е топло или ще вали сняг“.

Замествайки променливите в един съждителен израз със съждения, се получава съждение. За да се определи верността на едно сложно съждение, е необходимо да се знае каква е верността на съставящите го прости съждения и смисъла на свързващите ги логически операции.

Логическите оператори си имат **приоритет**, който определя последователността на тяхното изпълнение. С най-висок приоритет е отрицанието, следван от конкюнкция, дизюнкция, импликация и еквивалентност. Скобите се използват за промяна на приоритета на операторите.

Закони на съждителното смятане

Някои общовалидни правила за съждителното смятане са означени като закони.

1. ***Комутативен закон***
   p && q ⇔ q && p
   p || q ⇔ q || p
2. ***Асоциативен закон***
   (p && q) && r ⇔ p && (q && r)
   (p || q) || r ⇔ p || (q || r)
3. ***Дистрибутивен закон***
   (p || q) && r ⇔ p && r || q && r
   (p && q) || r ⇔ (p || r) && (q || r)
4. ***Закони на де Морган***
   !(p && q) ⇔ !p || !q
   !(p || q) ⇔ !p && !q
5. ***Закон за контрапозицията***
   p → q ⇔ !q → !p
6. ***Закон за изключеното трето***
   p || !p ⇔ 1
7. ***Закон за силогизма (транзитивност)***
   (p → q) && (q → p) ⇔ p ↔ q

Тези закони могат лесно да бъдат доказани чрез използване на таблици за истинност.

Пример: Да се докаже законът на де Морган !(p && q) ⇔ !p || !q

Първо е необходимо да се образуват всички възможни набори от стойности на двете логически променливи p и q. След това последователно се изчисляват и сравняват стойностите на левия и десния израз. Ако двата израза имат равни стойности за еднаквите стойности на променливите, можем да твърдим че законът е общовалиден.

p	**q**	**p &&q**	**!(p &&q)**	**!p**	**!q**	**!p || !q**
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

От законите са изведени и някои **следствия**:

1. ***Закон за слепване***
   p && q || p && !q ⇔ p
   (p || q) && (p || !q) ⇔ p
2. ***Закон за поглъщане***
   p || (p && q) ⇔ p
   p && (p || q) ⇔ p

3. ***Закон за съкращаване***

$p \| (!p \&\& q) \Leftrightarrow p \| q$

$p \&\& (!p \| q) \Leftrightarrow p \&\& q$

Законите на съждителното смятане и следствията могат да се използват за преобразуване на сложните изрази до по-прости.

Например: Да се опрости изразът p || (p && q).

```
p || (p && q) = (p && 1) || (p && q) = p && (1 || q) = p && 1 = p
```

Какво представляват алгоритмите

Програмите реализират алгоритми.

Алгоритъм е описание на последователност от краен брой действия, които се изпълняват за решаването на задача или клас от задачи. Задачите може да са математически или всякакви други. Например, за намирането на корените на квадратно уравнение по стандартната формула $x_{1,2}= (-b\pm\sqrt{D})/2a$, за приготвянето на ястие по дадена рецепта, за сглобяването на мебел, следвайки инструкциите на производителя и др. Действията, описани от алгоритъма трябва да са достатъчно прости и последователността на изпълнение на действията трябва да е точно определена, така че да могат да бъдат изпълнени от човек или машина без необходимост от осигуряване допълнителни инструкции.

Примерен алгоритъм за изчисляване на периметър и лице на правоъгълник по зададени дължини на две прилежащи страни е даден в следния пример:

```
1. Въвеждане дължината a на едната страна на правоъгълника.
2. Въвеждане дължината b на втората страна на правоъгълника.
3. Изчисляване P = 2 * (a + b).
4. Изчисляване S = a * b.
5. Извеждане стойността на P.
6. Извеждане стойността на S.
7. Край.
```

Компютърните алгоритми реализират модел за решението на задача, който зависи от знанията на програмистите, поставени специфични изисквания, възможностите на избрания език за програмиране и др.

Алгоритмите се прилагат върху конкретни **входни данни**. По време на изпълнение на алгоритъма, в общия случай се получават и обработват **междинни данни. Изходни данни** се наричат резултатните данни от цялостното изпълнение на алгоритъма.

Данните могат да са организирани по различни начини. Може да са конкретни числа и/или обекти; колекция (списък, множество или др.) с известен или неизвестен брой елементи; може да се четат от файлове или да се въвеждат от потребител по време на работа на програмата. Съответно, за решението на една задача може да се съставят много алгоритми и различни конкретни реализации.

Основни типове алгоритми са:

- ***линейни***, при които всички описани стъпки се изпълняват последователно, без значение от входните данни. Описаният по-горе алгоритъм е линеен.
- ***разклонени*** – в зависимост от конкретните входни данни и междинни резултати се изпълняват различни стъпки;
- ***циклични*** – част от стъпките на алгоритъма се изпълняват многократно, обикновено с различни стойности на участващите променливи;

- ***рекурсивни*** – при някоя от стъпките е необходимо да се изпълни същия алгоритъм директно (с други входни параметри) или индиректно (чрез друг алгоритъм).

Пример за разклонен алгоритъм е определянето на по-голямото от две числа a и b:

```
1. Въвеждане стойността на a.
2. Въвеждане стойността на b.
3. Проверка дали a е по-голямо от b. Ако a е по-голямо от b,
   преминаване на стъпка 4. В противен случай преминаване на
   стъпка 5.
4. Извеждане стойността на a. Преминаване на стъпка 8.
5. Проверка дали a е по-малко от b. Ако a е по-малко от b,
   преминаване на стъпка 6. В противен случай преминаване на
   стъпка 7.
6. Извеждане стойността на b. Преминаване на стъпка 8.
7. Извеждане низа „Двете числа са равни”.
8. Край.
```

В този пример, в стъпки 3 и 5 има условия, които определят различни начини, по които може да продължи изпълнението на алгоритъма.

Чрез цикличен алгоритъм може да изчислим сумата на числата от 1 до n (n>0).

```
1. Въвеждане стойността на числото n.
2. Проверка дали n > 0. Ако n > 0, преминаване към стъпка 3, в
   противен случай преминаване към стъпка 1.
3. Присвояване стойност 1 на променлива за брояч i.
4. Присвояване стойност 1 на променливата за сума S.
5. Проверка дали i < n. Ако i < n, преминаване на стъпка 6. В
   противен случай, преминаване към стъпка 8.
6. Изчисляване i = i + 1.
7. Изчисляване S = S + i. Преминаване към стъпка 5.
8. Извеждане стойността на S.
9. Край.
```

Стъпки 5, 6 и 7 може да се изпълняват многократно.

При рекурсивните алгоритми в някоя от стъпките при определено условие се обръщаме към същия алгоритъм, но с различни входни данни. Условията зависят от входните данни и трябва да са такива, че в определен случай рекурсивното използване да спре. Тривиален пример за рекурсивен алгоритъм е намирането на факториел от цяло неотрицателно число n. По дефиниция факториел от n се записва с формулата

```
n!=n*(n-1)*(n-2)...3*2*1, а
0! = 1
```

Тази дефиниция може да я представим и като рекурсивна

```
n!=n*(n-1)!,
0! = 1
```

т.е. за да намерим факториел от n трябва да умножим n с факториел от (n-1). При рекурсивните извиквания входният параметър ще намалява непрекъснато. Условието за

спиране на рекурсивните извиквания е входният параметър да е 0 или 1. Тази задача може да се реализира и чрез цикличен алгоритъм.

Създаване на алгоритми

Има различни начини за описание на алгоритмите – словесно, чрез блок-схеми и чрез програмни езици.

Словесно описаните алгоритми имат множество недостатъци – двусмислено тълкуване, не добра нагледност и невъзможност да бъдат изпълнени от машина.

Алгоритмите може да са представят графично чрез блок-схеми. Те използват стандартни блокове за означаването на отделни типове действия и стрелки, за указване на последователността на изпълнение на действията. Блок-схемите са по-разбираеми, лесно се четат, но не са подходящи за изпълнение от машина. Въпреки, че са удобни за описване и изучаване на по-елементарни алгоритми, те губят смисъла си и не се използват от опитни програмисти.

Описанието на алгоритъм на програмен език е най-високото ниво на формализация на алгоритмите. За да може да се опише алгоритъм на какъвто и да е език за програмиране, трябва добре да се познават:

- синтактичните възможности на езика;
- стандартни (и съответно ефективни) решения на основни задачи;
- библиотеки, свързани с решението на задачата – напр., може да е необходимо да се визуализира елементарен алгоритъм чрез сложни графични библиотеки, които може да бъдат използвани наготово;
- специализирани техники за разработка – напр. шаблони за дизайн;
- и др.

Познаването на един език за програмиране не е достатъчно за написването на алгоритъм. Езикът е само средство, чрез което трябва ефективно да се изрази намерено решение. При решаването на сложни задачи (за които е трудно да бъде намерено веднага компютърно решение), е подходящо да се започне с решаване „на хартия” – с описания, скици, диаграми и други. При това се извършват различни дейности като:

- описание на входни данни;
- очакван краен резултат;
- формално описание на известни решения – напр. чрез формули;
- търсене на неизвестни решения;
- измисляне на решение (ако не е намерено);
- избор на решение, ако има много такива;
- разделяне на логиката на решението в самостоятелно решими задачи, които след това се преобразуват в алгоритми.

В едно приложение може да е необходимо да се реализират много различни алгоритми. Реализацията на конкретен алгоритъм може да зависи от специфични изисквания, заложени при проектирането и дизайна на приложението – в самостоятелен метод, клас, интерфейс или др.

Сложност на алгоритъм

Често за решението на една задача може да се използват няколко различни алгоритъма. Някои алгоритми са по-ефективни от други. Оценката за ефективност е свързана с броя на изпълняваните операции върху входните данни, чийто брой се означава с N.

Сложност на алгоритъм е приблизителна оценка O на максималния брой операции, необходими за изпълнението на алгоритъм, зависеща от броя на входните данни N.

Сложността на алгоритъм се означава с главно O, а в скоби след него се задава приблизителния порядък на броя операции. Той е функция на входните данни, напр. O(N), O(N^2), O(N^3). Тази функция може да се опише и чрез конкретна формула, напр. O(N^2) = N^2+N/2+5. В общия случай обаче, е важен само порядъка („най-тежката" част от формулата), а не точната формула.

Има различни **видове сложност** на алгоритмите:

- **Константна** – означава се O(1). В този случай броят на операциите не зависи от броя на входните данни.
- **Линейна** O(N) – броят на операциите зависи линейно от броя на входните данни. Например, при намиране на сума (вж. описанието на алгоритъма за сумиране) на списък с N елемента, по N пъти се изпълняват действията „добавяне към сумата", „добавяне на 1 към брояча", „сравнение". Това прави 3*N броя основни операции, към които може да добавим и няколко константни действия. Тук зависимостта е линейна. Разбира се, това не е най-добрият алгоритъм за намиране на сума на числата от 1 до N. Алтернативен алгоритъм с константна сложност е намирането на сумата чрез аритметична прогресия по формулата ((1+N) /2)*N.
- **Квадратична** O(N^2) **–** при обем на входните данни N, ще се изпълнят операции с основен порядък N^2. Напр. ако за два списъка с числа от по M елемента (N=2*M), желаем да намерим произведенията на всеки елемент от единия списък с всеки елемент от другия списък, ще са необходими M^2 =$(N/2)^2$ броя основни операции.
- **Логаритмична** O(log(N)), O(N*log(N)) – това е относително ниска (между константна и квадратична) степен на сложност на алгоритъма, при която броят на входните данни N е равен на степен на броя на операциите. По дефиниция числото x = $\log_a b$ (логаритъм от b при основа a) е такова, че a^x=b. При изчисленията на сложността на алгоритъм се използват различни основи (2, неперовото число e ≈ 2.718 или др.), които може да не се означават. Например зависимостта между O и N при основа 2 е O=$\log_2$N или N=$2^{O.}$ Пример за логаритмична сложност имаме при търсене в подреден списък с N числа. За да намерим числото x, не е необходимо да прочетем всички елементи от списъка. Един възможен алгоритъм е да се сравни x със средния елемент; в зависимост от резултата от сравнението или е намерен търсения елемент, или търсенето може да продължи по-подобен начин, но само в едната част от списъка. Така максималният брой на операциите ще бъде броят на разделянията на непрекъснато смаляващия се списък. Напр., при 10 елемента, първо ще смалим списъка до 5, след това до 2 или 3 елемента и накрая до 1. Така ще имаме общо 3 сравнения (действия) т. е. 2^3=8≈10 т. е. сложността е логаритмична. При 1000 елемента максималният брой сравнения ще е приблизително 10 (2^{10}=1024).
- **Експоненциална** сложност има, когато броят на основните операциите се увеличава стремително при увеличение на броя на входните данни. Такива зависимости са O(N^K), O(K^N), O(N!) и изглеждат плашещи при по-големи стойности на N и K. (Квадратичната сложност е не толкова страшен случай на N^K.)
- **Сложност, зависеща от няколко променливи** – ако имаме два списъка с различен брой M и N на елементите, то всевъзможните двойки комбинации от елементи от двата списъка е M*N. Тогава сложността на алгоритъм, извеждащ всевъзможните двойки е O(M*N).

Сравнението на сложностите на няколко програмни реализации на един и същи алгоритъм понякога е трудна задача. Ако всяка реализация използва различни готови или създадени от програмиста елементарни операции (методи), ще трябва да се оценяват сложностите и на елементарните операции. Друг начин е да се сравнят времената за изпълнение върху едни и същи набори от входни данни, при еднакви условия. В този случай трябва да се подберат подходящи набори от входни данни, защото някои

реализации може да са много по-добри при относително малък брой входни данни, а други – при по-голям.

Въпроси и задачи за упражнения

1. Да се запишат в двоичен код десетичните числа: 19, 34 и 55.
2. Да се запишат в осмичен код десетичните числа: 67, 93 и 321.
3. Да се запишат в шестнайсетичен код десетичните числа: 1234, 3123 и 62128.
4. Да се намери в десетичен код стойността на числата: $111001_{(2)}$, $276_{(8)}$, $A3D_{(16)}$.
5. Да се запишат в двоична бройна система осмичното число 2471 и в осмична бройна система двоичното число 11001010101110.
6. Да се запишат в осмична бройна система шестнадесетичното число AB9 и в шестнайсетична бройна система осмичното число 486.
7. Кои от следните изречения са съждения:
   - Две плюс две е равно на четири.
   - Навън вали ли дъжд?
   - Земята е кръгла и се върти около собствената си ос.
   - Ако a = b и b = c, следва че a ≠ c.
   - Днес ли е Коледа?
   - Омръзна ми да гледам телевизия!
8. Напишете таблиците за истинност на следните изрази:
   - (p || q) && !q
   - (p → q) → r
   - q ↔ (!p || !q)
   - (p || q) ↓ (p && q)
   - (p || !q) ↔ (!p && q)
9. Докажете с таблица за истинност следните закони:
   - (p || q) || r ⇔ p || (q || r)
   - !(p || q) ⇔ !p && !q
   - p →q ⇔!q → !p
   - (p && q) || r ⇔ (p || r) && (q || r)
   - (p → q) && (q → p) ⇔ p ↔q
10. Да се опростят чрез еквивалентни преобразувания следните изрази:
    - !(p → q) → !q
    - (!p && (p || q)) → q
    - ((p → q) && (q → r)) → (p → r)
    - (p && (p → q)) → q
    - ((p || q) && (p → r) && (q → r)) → r
11. Да се състави алгоритъм за изчисляване на лице на трапец по зададени дължини на двете основи и височината на трапеца.
12. В група от N човека, M са мъже (N>M). Колко процента са жените? Да се състави алгоритъм и за изчисляване на търсения процент жени.
13. Дадена е редица от стойности A1,A2…An, където n е естествено число. Да се опише с текст цикличен алгоритъм за намиране най-малкия елемент в редицата, както и неговия пореден номер.

Глава 3. Основни понятия в езиците за програмиране. Java

Езиците за програмиране, подобно на естествените езици имат основни елементи – символи, различни видове думи (стандартни ключови думи, оператори, разделители, идентификатори) и синтаксис, определящ правила за съставяне на имена, изрази, конструкции, програмни части (блокове, методи) и като цяло на кода на програмите.

В кода на програмите на Java е позволено използването на UNICODE символи. Например, за имена на променливи и методи може да се използват символи на кирилица. В езика се прави разлика между малки и главни букви. Напр., int е стандартна ключова дума, а Int – не.

В тази глава ще разгледаме основните понятия и синтаксиса на някои езикови конструкции в Java. Част от тях ще са представени по-подробно в следващи глави.

Ключови думи в Java

Запазените ключовите думи в Java се състоят от ASCII символи и са описани в следната таблица:

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

Легенда:

* – не се използват (включени са, тъй като се използват в повечето езици за програмиране)

** – добавени в JDK 1.2

*** – добавени в JDK 1.4

**** – добавени в JDK 5.0

Ключовите думи условно са разделени в няколко категории:

Категория	**Ключови думи**
модификатори за достъп	private, protected, public
модификатори за класове, методи и променливи	abstract, class, extends, final, implements, interface, native, new, static, strictfp, synchronized, transient, volatile
контрол на изпълнението	break, case, continue, default, do, else, for, if, instanceof, return, switch, while
контрол на пакетите	import, package
примитивни типове	boolean, byte, char, double, float, int, long, short
прехващане на грешки	assert, catch, finally, throw, throws, try
изброяване	enum
други	super, this, void
неизползвани	const, goto

Бели полета

Белите полета са символи, които служат за форматиране на кода и разделяне на думите една от друга. Те са два основни вида:

- **Край на ред** (Line Terminators) – използват се ASCII символите LF (с код 10 – newline), CR (с код 13 – return) или комбинация от двата. В зависимост от текстовия редактор и операционната система при натискане на клавиша „Enter" се записва някой от вариантите.
- **Бяло поле** (White Space) – интервал (код 32) и таб (код 9).

Разделители

Използват се за разделяне на специални части от кода, както следва:

- ( ) – заграждат формалните и фактическите параметри при деклариране и извикване на методи; участват в математически изрази; използват се при преобразуване по тип и при оператори за контрол на изпълнението на кода;
- { } – при работа с масиви и дефиниране на блокове;
- [ ] – при работа с масиви;
- <> – при работа с шаблонни типове (generics)
- ; – край на команда;
- , – разделя променливи при декларация;
- . – за достъп до елемент на клас/обект и при описание на под-пакети;
- : – при работа с някои видове цикли;
- ... – дефиниране на променлив брой параметри;
- @ – определя началото на анотация.

Коментари

Коментарите описват в свободен текст помощна информация с цел по-добро разбиране на кода, и/или използването му от други потребители. Те са три вида:

- Започват с // – коментар на един ред;
- Започват с /* и завършват с */ – коментар на няколко реда;
- Започват с /** и завършват с */ – коментар на няколко реда, който се използва за автоматично генериране на HTML-документация с инструмента javadoc. В нея може да се задава информация (за автор, клас, метод, параметри и др.) чрез специфични тагове.

Подробна информация за правилата за писане на javadoc-документация може да намерите на адрес http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

Литерали

Литералите представят числови, булеви и други стойности от различни типове:

- Число – описани по стандартни начини чрез цифри и други символи;
- Низ – текст, заграден в кавички;
- Символ – символ, заграден в апострофи;
- true, false – литерали за булеви стойности;
- null – литерал за описание на динамичен обект, който не е създаден.

Оператори и изрази

Операторите представят действия върху различни типове данни – числа, низове и др. Някои от операторите са ключови думи, а други са разделители. В следващата таблица са представени различните видове оператори.

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

Данните, участващи в операциите, се наричат операнди или аргументи.

Има оператори с един, два или три аргумента. Те се наричат съответно унарни (unary), бинарни (binary), тройни (ternary).

Израз е комбинация от оператори и операнди. Резултатът от изчислението на израз е стойност от някакъв тип – за число, низ или друг. Операндите могат да бъдат както конкретни литерали, така и променливи, константи или функции, получаващи стойност по време на изпълнение на програмата.

Начинът на изчисляване на един израз зависи от **приоритета** и **асоциативността** на участващите оператори.

Приоритетът определя реда на прилагане на операторите. Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Например, при аритметичните оператори, с по-висок приоритет са *, /, %, а с по-нисък + и -. Приоритетът може да се промени като се използват скоби.

В следната таблица може да се види резултатът от изчислението на елементарен израз, с и без използване на скоби.

№	Израз	Резултат
1.	1+2*3	7
2.	(1+2)*3	9
3.	1+(2*3)	7

Изрази 1 и 3 са еквивалентни. В зависимост от конкретната задача, която трябва да се реши, всеки един от изразите може да е правилен.

Ако не сме сигурни в приоритетите на операторите е добре да използваме скоби, за да не възникват семантични грешки.

Асоциативността определя реда на присъединяване на операндите към оператора. При повечето оператори, първо се взима стойността на левия операнд, а след това на десния. Те се наричат ляво-асоциативни. При унарните оператори, обаче, има само един десен операнд и първо се изчислява неговата стойност. Такава е логиката и при оператора за присвояване. Първо се изчислява изразът в дясно, а след това той се присвоява на променливата в ляво. Унарните оператори и операторите за присвояване са дясно-асоциативни.

Идентификатори

Идентификаторите задават уникални имена на променливи, класове, методи и др. Те са уникални в определена област на действие, напр. пакет, клас, метод, блок. Имената не може да съдържат оператори, разделители и бели полета и не могат да започват с цифра. Идентификатори не могат да бъдат ключови думи, оператори, литерали.

Величини

Величините служат за съхранение на стойности от определени типове. Имат три основни характеристики – **тип, име/идентификатор (за някои видове величини) и стойност.** Величините са два вида:

- **Променливи** – могат да променят стойността си по време на изпълнение на програмата. Имат име, чрез което става обръщение към тях. Биват два вида:
  - **Вътрешни** – контролират се от програмата;
  - **Външни** – стойността им се определя от средата, в която се изпълнява програмата. Напр. има променлива за системното време.
- **Константи** – стойността им се задава еднократно. Те са два вида:
  - **Именовани константи** – подобно на променливите – имат име, но не могат да променят стойността си;
  - **Литерали** – записана като текст стойност, която няма име.

В зависимост от това, къде се използват величините, могат да имат специфични наименования – поле на клас, променлива, константа, формален параметър на метод, фактически параметър на метод.

Типове

Типът определя дефиниционна област на величините, т.е. множество от възможни стойности, които могат да се присвояват на величините. Всеки тип си има име и определя размер на паметта (брой битове), необходими за запис на величина от съответния тип.

В Java съществуват **два вида типове – примитивни** (прости, елементарни) и **сложни** (съставни)**.**

- ***Примитивните*** типове са четири вида:
  - за цели числа – ***byte, short, int, long;***
  - реални числа – ***float, double;***
  - булеви стойности – ***boolean;***
  - символи – ***char***.
- ***Сложните*** типове обикновено комбинират няколко величини от различни типове и/или методи . Сложни типове в Java се реализират чрез:
  - ***масиви;***
  - ***класове;***
  - ***интерфейси;***
  - ***изброим тип*** *(който е частен случай на клас)*.

Специален сложен тип в Java е класът ***String*** за работа с низове. Той донякъде прилича на примитивен тип, тъй като за него са създадени специални оператори за присвояване на стойност (=) и за конкатениране (слепване) на низове (+).

Декларация и инициализация на променлива в Java

Променлива в Java се декларира по следния начин:

```
<тип> <идентификатор>;
```

Задаването на стойност на променлива се нарича инициализация и се извършва с оператора за присвояване „=”.

Например,

```
double w;       // Декларация на променлива с име w от тип double
w = 7.3;        // Задаване на стойност на променливата w
byte age = 1; // Декларация и инициализация на променлива от тип byte
String name = "Иван"; // Променлива от тип низ
w = w + 2;      // На променливата w, която е ляв операнд за оператора =,
                // се присвоява стойността на израза от дясно.
```

При израза „w = w+2” първо се изчислява десният израз – текущата стойност на w се сумира с числото две, и след това полученият резултат се присвоява на операнда в ляво. По този начин променливата w получава нова стойност, равна на текущата стойност, увеличена с 2. Ако впоследствие променливата w се използва в някакъв израз, тя ще участва с новата си текуща стойност. Разбира се, в този пример се използва и оператор „+” за сумиране на числа.

Освен променливи, в примера са използвани и литерали – 7.3, 1, „Иван”, 2.

В следващия пример сме създали сложен израз за конкатениране на няколко низа което се записва със символа на аритметичния оператор за сумиране „+”.

```
System.out.println(name + " е на " + age + " г. и тежи " + w + " кг.");
```

В случаите, в които в израза участват низове се подразбира, че операторът „+” е операторът за слепване на низове. Резултатът от слепването е низ, като се взимат стойностите на променливите и се преобразуват до низ, ако е необходимо.

Резултатът от изпълнението на горния код е низът:

```
Иван е на 1 г. и тежи 9.3 кг.
```

При декларация на променливи, може да се декларират едновременно няколко променливи от един тип, разделени със запетаи:

```
int i=1, j, k;
```

Константи в Java

Константи се декларират подобно на променливи, но с модификатор final пред типа. Задължително при декларацията се задава стойност на константата и тази стойност не може да бъде променяна при следваща операция.

Например,

```
final double PI = 3.14;
```

Ако константата се декларира като поле на клас, може да ѝ се зададе и модификатор static. Не-статичните (т.е. динамичните) променливи и константи се създават (за тях се заделя памет) за всеки конкретен обект от класа, а статичните се създават еднократно.

```
static final double PI = 3.14;
```

Конструкции за контрол на изпълнението

В кода на програмите се описват последователно (линейно) различни команди. Изпълнението им обаче, в общия случай не е последователно. За да променим последователността на изпълнение на кода използваме конструкции (оператори) за контрол на изпълнението:

- Чрез **условни оператори if, if-else**, и **оператор за избор от варианти switch-case** се изпълнява код само при определени условия;
- Чрез **оператори за цикъл while, do-while и for** даден код се изпълнява многократно.

Методи

В методите се реализират алгоритми, като се използват данни (във вид на величини, обикновено променливи), върху които се прилагат различни действия – оператори, конструкции за контрол на изпълнението и други. При изпълнението си, един метод може да „извиква" други методи.

При реализацията на алгоритмите, методите могат да използват променливи и константи, декларирани на различни нива – като полета на клас, параметри на методи, локално дефинирани променливи.

Методите могат да бъдат извиквани многократно с различни стойности на параметрите, при което резултатите от изпълнението им в общия случай също са различни.

Методите имат декларативна част и дефиниция – тяло, в което се описва кодът на метода.

Някои методи може да връщат стойност. Методи, които връщат стойност се наричат функции, а тези, които не връщат стойност, се наричат процедури.

Основният вид на декларацията на метод е:

```
   [<модификатори>] [<тип на връщана стойност>] <име на метод>([<списък с формални параметри>])
```

В тези неформални описания на синтаксиса, правоъгълните скоби ([]) описват незадължителни елементи, а ъгловите скоби (<>) – неща, които трябва да се задават.

Модификаторите не са задължителни. Те са два вида:

- **модификатори за достъп** – определят различни видове видимост в класовете и пакетите и са четири вида – public, private, protected, по-подразбиране (ако не е зададен);
- **други модификатори** – засега ще използваме само модификатора static, който указва, че методът е статичен и може да се използва, без да е необходимо да се създава обект на класа.

Типът на връщана стойност може да е:

- **произволен тип** – както примитивен, така и сложен. В този случай в тялото на метода трябва да има оператор return, който да връща стойност от съответния тип ;
- **void** – ключовата дума void указва, че методът не връща стойност.

Когато методът е конструктор, не се задава тип на връщана стойност.

Списъкът с формални параметри съдържа декларации на променливи, разделени със запетая и не е задължителен.

Името на метода и списъка с формални параметри трябва да са уникални в рамките на текущия клас. В един клас може да съществуват методи с еднакви имена, но с различен брой и/или тип параметри.

Тялото на метода се описва в блок от команди, ограден с отваряща и затваряща фигурни скоби.

За да предизвикаме изпълнение на метод, трябва да се обърнем към него (да го извикаме) с името му, като зададем списък с конкретни/фактически параметри, съответни по тип на декларираните за метода формални параметри. При всяко изпълнение на метода, формалните параметри се инициализират със стойностите на съответните (според позицията си) фактически параметри. Фактическите параметри могат да бъдат произволни изрази, напр., литерали, променливи или методи. Важното е да са от същия (или съвместим) тип, от който е съответният формален параметър

В тялото на един метод може да използват само описаните в декларацията му формални параметри, локалните променливи, както и общите за класа полета.

В следващия пример е създаден метод s, който изчислява лице на квадрат по зададена като формален параметър страна a, и извежда резултата в конзолата. Методът s е извикан в основния-main метод, от който започва изпълнението на програмата.

```
public class Square {
      public static void s(double a){
            double s; // Декларираме променлива s за лицето
            s = a*a;
            System.out.println("Лицето на квадрат със страна "
                                          + a + " е " + s);
      }
      public static void main(String[] args) {
            s(2.1);
            s(3);
      }
}

Резултат:
Лицето на квадрат със страна 2.1 е 4.41
Лицето на квадрат със страна 3.0 е 9.0
```

Методът s може да бъде реализиран като функция.

```
public class Square2 {
      public static double s(double a){
            return a*a; // С return връщаме стойност
                        // от декларирания за метода тип
      }
      public static void main(String[] args) {
            double a = 4;
            double s = s(a);
            System.out.println("Лицето на квадрат със страна "
                        + a + " е " + s);

            a = 5;
            System.out.println("Лицето на квадрат със страна "
                        + a + " е " + s(a));
      }
   }
   Резултат:
   Лицето на квадрат със страна 4.0 е 16.0
   Лицето на квадрат със страна 5.0 е 25.0
```

Резултатът от изпълнението на функция може да се присвои на променлива от подходящ тип – `s = s(a)`. Той може директно да участва в израз, както е показано при

второто извеждане на информация в конзолата. В Java е разрешено името на променлива да съвпада с името на метод.

Кой вариант на решение на задачата е по-добър?

От една страна, при първото решение няма дублиране на код за извеждане в конзолата, а във второто – има, което може да предизвика възникването на случайни грешки (при копиране). Кодът за отпечатване обаче може да се изнесе в отделен метод, притежаващ параметри за страна и лице.

От друга страна, първото решение е тясно свързано с извеждането на резултата, т.е. при него методът за лице може да се използва само за конзолни приложения. По този начин ограничаваме възможностите за многократно използване на метода и ако желаем да направим например графично или уеб приложение, ще трябва да пренаписваме кода. За елементарен метод пренаписването не е проблем, но за по-сложен – може да се наложат множество промени.

Ако се създават библиотеки с класове и методи, които да се използват от други потребители, не е удачно методите, реализиращи работната (бизнес) логика да извеждат информация в конзолата.

Масиви

Масивите описват списък от еднотипни елементи. Всеки елемент има индекс, който е число. В Java индексите са последователни цели числа, като първият елемент има индекс 0, следващия 1 и т.н.

Масивът е променлива и се декларира по подобен начин. Тук правоъгълните скоби са задължителна част от синтаксиса за деклариране.

```
<тип> [] <име на променлива>;
```

Например,

```
int [] intArray;
double [] doubleArray;
```

След като сме задали стойности на елементите, може да се обръщаме към тях с името на променливата за масив и индекса:

```
intArray[0]=1;
doubleArray[1]=1.1;
```

Подробности за инициализация на елементите и работа с масивите ще разгледаме в глава 8.

Класове

От синтактична гледна точка класът описва съвкупност от полета (декларации на променливи и константи) и методи. В рамките на класа, методите му могат да използват (виждат) полетата и методите на същия клас, но не и методи и полета, описани в други класове. Достъпът до не-собствени полета и методи става индиректно, чрез клас или обект.

По-рядко използвана техника е като членове на класа, освен полета и методи, да се описват вложени класове.

Тялото на класа, подобно на методите се огражда във фигурни скоби, а декларацията на клас е от вида:

```
[<модификатори>] <име на клас> [extends <име на клас-родител>]
```

Като модификатори за достъп за обикновените (не-вложени) класове могат да се задават само public или по подразбиране (нищо), а за вложените – всички.

След ключовата дума extends може да се задава клас-родител, който се наследява от текущия клас. Ако не е указано наследяване, автоматично се наследява класът Object от пакета java.lang.

В Java може да се наследява само и задължително един родителски клас. По този начин се реализира единично наследяване, и не се допуска множествено наследяване на класове. Поради това всички класове в Java образуват обща йерархия, с корен класът java.lang.Object. В езика C++ също са реализирани единично и множествено наследяване, но при това не е задължително да се указва родителски клас т.е. класовете са организирани в множество отделни йерархии.

В Java множествено наследяване се организира чрез използването на интерфейси.

Интерфейси

Основни недостатъци на множественото наследяване на класове са двусмислията, които възникват при използването на родителски полета и методи с еднакви декларации. Това създава проблеми, както при създаване на инструментите за компилиране така и за програмистите.

Чрез интерфейсите в Java се предлага улеснен подход за реализация на множествено наследяване.

Синтаксисът за декларацията на интерфейс е:

```
[<модификатори>] <име на интерфейс> [extends <списък с имена на интерфейси-родители>]
```

При наследяване може да се зададат множество интерфейси-родители, разделени помежду си със запетая.

Интерфейсите, подобно на класовете могат да съдържат полета и методи, но само константи и декларации на методи. Например, може да създадем интерфейс за обекти, притежаващи повърхнина.

```
public interface Surface {
      final static double PI = 3.14;

      public abstract double s();// метод за лице
   }
```

Изходният код на интерфейсите, както и на класовете, се записва във файл с разширение ".java", а при компилирането му създава съответен class-файл.

Методи, които имат само декларация в даден клас или интерфейс, са абстрактни и се означават с модификатор abstract.

При интерфейсите не е задължително да се указва, че методът е абстрактен, а също така не е задължително полетата да се описват като константи (с final static), защото по подразбиране се приема, че са такива.

От версия 8 на Java в интерфейсите може да се създават методи, декларирани с модификатор default, както и static методи. И при двата вида може да се задава тяло на метода, което приближава Java до езиците за програмиране, притежаващи истинско множествено наследяване.

Методите на интерфейса се реализират в класове, които го имплементират (implement-осъществявам, реализирам).

Имплементация на интерфейси

За да се укаже, че един клас „наследява" интерфейс, се използва разширен синтаксис за декларация на клас:

```
   [<модификатори>] <име на клас> [extends <име на клас-родител>] [implements
<списък с интерфейси>]
```

При това има две възможности за класа-наследник. Едната възможност е в тялото на класа да се дефинират всички методи, описани в интерфейса. Другата възможност е класът да се декларира като абстрактен, при което реализацията на наследените от интерфейса методи да остане за неговите наследници.

Например, различни класове за описание на геометрични фигури може да имплементират интерфейса Surface – Квадрат, Правоъгълник, Окръжност, Куб, Сфера и др. Въпрос на преценка на проектанта е каква точно йерархия от наследявания между самите класове е подходящо да създаде. Всеки от класовете обаче, може да има собствена реализация на метода s(). Някои класове може да използват константата PI, дефинирана в интерфейса, а други – не.

```
// Клас за окръжност
public class Cirlce implements Surface{
      public double r; // радиус

      // реализация на метода на Surface
      public double s(){
            return PI*r*r;
      }
}

// Клас за правоъгълник
public class Rectangle implements Surface{
      public double a, b; // страни

      // реализация на метода на Surface
      public double s(){
            return a*b;
      }
}
```

Модификатори за достъп

В тялото на класа/интерфейса се описват полета и методи, които имат различни нива за достъп (видимост). Видимостта се специфицира чрез следните модификатори:

- **public** – видими са от всички останали класове
- **без модификатор (по подразбиране)** – видими са в само в класовете от пакета, в който е създаден текущият клас;
- **protected** – видими са само в текущия клас и наследниците му;
- **private** – видими са само в текущия клас.

Блокове

В Java, кода може да се описва в различни блокове, оградени с фигурни скоби: блок за тялото на клас; блок за тялото на метод, вложен в тялото на класа; блокове за операторите за контрол на изпълнението, които пък се влагат в методите и др.

Основно правило е, че **променливите и методите са видими в блока, в който са дефинирани и в неговите вложени блокове.**

В тялото на клас и в метод може да има дефинирани променливи с еднакви имена – съответно поле и локална променлива или формален параметър. Дефинираната във вложения блок променлива припокрива полето и донякъде то става „невидимо" за метода. В този случай, за да се обърнем към полето на класа (например var) може да използваме пълното му име – this.var. Ключовата дума this е специална променлива-референция към текущия обект. Тя се инициализира и започва да сочи към текущия обект по време на създаване на обекта.

В метод може да се дублират имена на променливи, само ако те не се припокриват т.е. ако са в различна йерархия от блокове. В частност, не може да се дублират имена на локална променлива и формален параметър или локална променлива на метод в основния блок и такава, дефинирана във вложен за метода блок.

Конструктор на клас

Конструкторите са методи на класа, чрез които се създават обекти на класа. При стартиране на конструктор, виртуалната машина на Java заделя памет за полетата на класа и ги инициализира с определени начални стойности. Една от основните идеи на конструктора е да зададе начални стойности на полетата с помощта на параметрите.

За конструкторите може да се изброят няколко основни характеристики:

- Конструкторът е специален метод на класа;
- Името на конструктор съвпада с името на класа;
- За конструктор не се указва тип на връщана стойност;
- В един клас може да има няколко конструктора – с различни по тип и/или брой параметри (имената на конструкторите са еднакви);
- Ако не е описан конструктор, в класа съществува конструктор по подразбиране без параметри (той се създава автоматично). Ако е описан конструктор с параметри, не се създава автоматично конструктор по подразбиране (и трябва да бъде създаден изрично, ако е необходим);
- Конструктор се извиква винаги при създаване на обект от клас;
- При извикване на конструктор, ВМ на Java заделя динамична памет за обект и за всички негови полета (за някои от полетата на класа също може да се извика конструктор);
- В конструкторите обикновено се задават първоначални стойности за атрибутите. Ако не се зададат стойности, атрибутите се инициализират автоматично със стойности по подразбиране (0 за числа, null за обекти, false – за boolean);
- Може да има конструктори, които са с модификатор за достъп private и са недостъпни. При това е необходимо да се осигури друг начин за достъп до членовете на класа – чрез статични полета и/или статични Factory-методи (за които обикновено се използват стандартни имена, като напр. getInstance).

В следващия пример е разширен класът за човек, представен в първа глава.

Добавен е конструктор за инициализиране на полетата на класа. Пренаписан (override) е един от стандартните методи, наследени от класа Object – String toString(), чиято идея е да връща низ, съдържащ подробна информация за конкретен обект на класа.

```
public class Person {
	private String name;
	private int age;

	// Конструктор - задава първоначални стойности на полетата на класа
	// чрез формалните параметри
	public Person(String name, int age){
		this.name = name;
		this.age = age;
	}

	// Метод, връщащ низ с информация за конкретен човек
	public String toString(){
		return name + " е на " + age + " г.";
	}
	public void setAge(int age){
		this.age = age;
	}
}
```

Полетата на класа са недостъпни чрез обект на класа, защото са зададени с модификатор за достъп private. Промяната на стойностите им може да бъде осъществена единствено чрез конструктор или чрез методи като setAge().

Добрият стил на програмиране изисква при деклариране на конструктора, имената на формалните параметри да съвпадат с имената на полетата, които инициализират. Поради това към полетата се използва обръщение с пълните им имена, включващи this. Тази техника улеснява процеса на кодирането, а впоследствие разчитането на кода и откриването на грешки. Неприятна би била ситуацията, ако в класа има още няколко полета за низове и формални параметри, означени с s1, s2, … sn. Тогава ще е трудно да се определи, например, кой от параметрите се отнася за полето name на човек?!

Обекти

Обектите се създават с помощта на оператора new, последван от обръщение към конструктор и задаване на фактически параметри, съответни на формалните:

```
new <конструктор>(<списък с фактически параметри>)
```

Създаденият динамичен обект може да се свърже с променлива-референция, декларирана с тип-класа на обекта или с тип на родителски клас или интерфейс.

Обръщението към полетата и методите на обекта се извършва като след променливата се запише оператор ‘.’ (точка) и името на желаното поле/метод.

Чрез променлива, декларирана с тип на родителски клас/интерфейс, може да се обръщаме само към методи, дефинирани и/или декларирани в родителя.

```
public class TestPerson {
	public static void main(String[] args) {
		// Създаване на обекти от клас Person
		Person p1 = new Person("Иван", 2);
		Person p2 = new Person("Мария", 3);

		// Извеждане на информация за хората...
		// ... с явно извикване на метод toString()
		System.out.println(p1.toString());
		// ... с автоматично (неявно) извикване на метод toString()
		System.out.println(p2);
```

```
        }
}

Резултат:
Иван е на 2 г.
Мария е на 3 г.
```

Методът toString() е наследен от java.lang.Object. Стандартната логика, която той трябва да реализира, е да връща низ, с описание на текущия обект. Тъй като методът и логиката му са известни, той често се използва автоматично. Поради тази причина, в примера, той е използван автоматично и неявно при извикването на метода System.out.println(p2).

Инстанция

Конкретен представител на клас или интерфейс, се нарича инстанция (instance).

Всеки обект е инстанция на класа, чрез който е създаден. Обектът е инстанция **и на всички родителски класове и интерфейси**, защото притежава и техните характеристики (полета и методи).

Например, обект от клас Circle е инстанция на Circle, Surface и Object.

Пакети

Пакетите обединяват класове и интерфейси, между които има някаква логическа връзка. Например класовете, описани в настоящата глава 3, е подходящо да бъдат обединени в пакет с име chapter3.

Всеки пакет може да има нула или повече под-пакети – например под-пакет interfaces за пакет chapter3.

Пълното име на пакет включва имената на всички родителски пакети с разделител '.' (точка). При това, първо се задава името на основния пакет, след това под-пакет и т.н. Например, chapter3.intrerfaces, chapter3.person и т.н.

Принадлежност на клас/интерфейс към пакет се декларира преди описанието на класа:

```
package <пълно име на пакет>;
```

Понятието пакет съответства на понятието директория във файловата система на една операционна система. За да се компилира един клас, трябва java-файлът да се намира в съответната на пакета йерархия от директории. При компилиране class-файловете се записват в също такава йерархия от директории.

За да се използва клас от друг пакет, е необходимо да се включи с клауза import. Тя се записва след декларацията на пакета, ако е зададена такава и преди описанието на класа. Изключение правят класовете от пакета java.lang, които са често използвани и не е необходимо да се включват изрично.

С маска '*' може да се включат всички класове от даден пакет по следния начин „import име_на_пакет.*;"

В следващия пример за работа с календар са включени три класа с import.

```
import java.util.GregorianCalendar;
import java.util.Locale;
import java.text.DateFormat;

public class CalendarTest {
```

```
      public static void main(String[] args) {
            // Създаване на обект за текущата дата
            GregorianCalendar gc = new GregorianCalendar();

            // Отпечатване на датата
            System.out.println(gc.getTime());

            // Форматиране на датата с локализация на български
            String bgFormat;
            bgFormat = DateFormat.getDateInstance(DateFormat.FULL, new
Locale("bg", "bg")).format(gc.getTime());
            System.out.println(bgFormat);
            bgFormat = DateFormat.getDateInstance(DateFormat.MEDIUM, new
Locale("bg", "bg")).format(gc.getTime());
            System.out.println(bgFormat);
      }
}
Примерен резултат:
Fri Nov 27 09:45:54 EET 2015
27 Ноември 2015, Петък
27.11.2015
```

За да разберете в детайли реализацията, моля, прочетете самостоятелно документацията на използваните класове или разгледайте примера отново на по-късен етап!

Програма в Java

Програма е клас, притежаващ специален метод, с декларация „public static void main(String [] args)”. Изпълнението на програмата започва винаги от този статичен метод.

Статични и динамични методи и полета

За да се изпълни статичен метод (описан с модификатор static), не е необходимо да се създава обект на класа.

Динамичните методи (без модификатор static) могат да се изпълняват само за/чрез обект на класа.

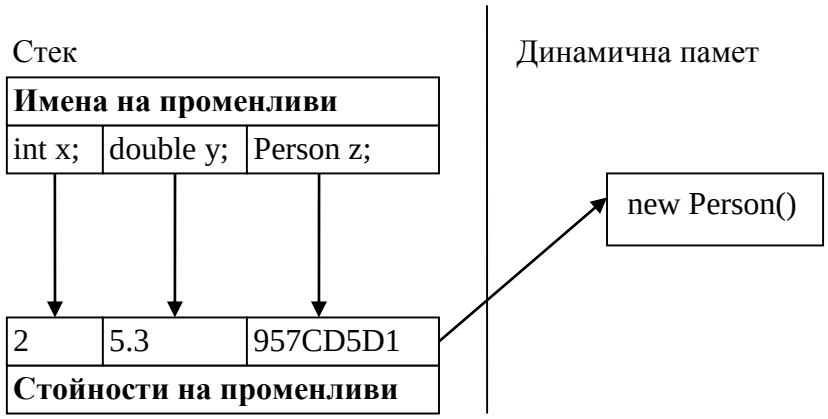
Може да се каже, че **статичните полета и методи са елементи на класа**, а **динамичните – са елементи на обекта**.

Към динамични полета и методи се обръщаме чрез име на променлива на обект **и оператор ‘.’**, а **към статични – чрез име на клас и оператор ‘.’**;

Статичен метод не може да използва директно не-статични полета и методи, дори ако те са описани в същия клас. За да се направи това, трябва да се създаде обект на класа и да се използва него.

Променливи в стека и динамичната памет

Имената на променливите се записват в стека на програмата. Стойностите на величините, в частност и стойностите на променливите, също се записват в стека, но на отделно място. При обръщение към променлива се взима съответната ѝ стойност.



Фиг. 3.1. Опростена представа за запис на променливи в паметта

Динамичните обекти (създадени с оператор new) се записват на някакъв адрес **в динамичната памет**.

Стойностите на променливите от прости типове и литералите се записват в стека. Стойностите на променливите от съставни типове не се записват директно в стека. Вместо това, в стека се записва указател към обекта в динамичната памет.

В Java променливите за динамични обекти се наричат референции (references – препратки), защото реферират (сочат) към определена част от динамичната памет.

Докато променливите за прости типове са твърдо свързани с адреси от статичната памет (стека), то променливите-референции може по време на изпълнението на програмата да реферират различни динамични обекти (чрез различни адреси).

Преди задаване на стойност на динамичната променлива, тя сочи към нулевия адрес, означен с null в Java.

Работата с адреси в Java става невидимо за програмиста, докато в други езици за програмиране, напр. в C++, има предвидени специални механизми – указатели и псевдоними, които се задават чрез синтактични конструкции.

Предаване (присвояване) на параметри по адрес и по стойност

В езиците за програмиране има два основни начина за предаване на параметри към методи:

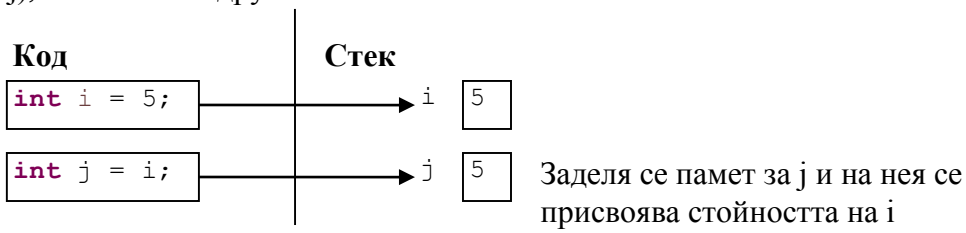
- **по стойност** – формалните и фактическите параметри са независими;
- **по адрес** – формалните и фактическите параметри са зависими, ако променим в метод формалния параметър, ще трябва да се промени и фактическия.

При предаването, формалните параметри получават стойността на фактическия параметър. Може да се каже, че **предаването на параметри е вид присвояване**.

Величините от примитивен тип в Java се предават/присвояват по стойност.

```
int i = 5;
int j = i; // присвояване по стойност
           // i и j са с различни собствени памети за стойностите
j++;
System.out.println("i=" + i);
System.out.println("j=" + j);

Резултат:
i=5
j=6
```

За всяка от променливите i и j в стека има самостоятелна памет за имената и стойностите им (фиг. 3.2) . Промяната на стойността на коя да е от променливите (в случая j), не повлиява другата.



Фиг. 3.2. Присвояване/предаване по стойност

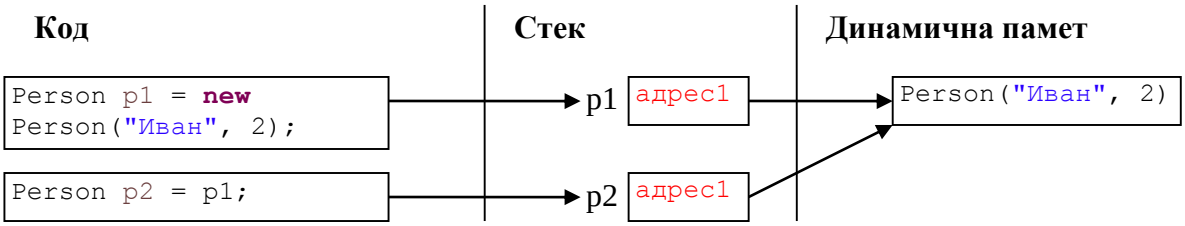
Величините от сложни типове (обекти и масиви) се предават/присвояват по адрес.

```
Person p1 = new Person("Иван", 2);
Person p2 = p1; // присвояване по адрес
                // p1 и p2 са референции към един и същи динамичен обект
p2.setAge(3);   // промяна на p2
System.out.println(p1); // отпечатване на p1
System.out.println(p2); // отпечатване на p2

Резултат:
Иван е на 3 г.
Иван е на 3 г.
```

Когато на една променлива-референция се присвоява друга променлива, то на първата се присвоява само адреса на втората, без да се създава нов динамичен обект. Това, всъщност е отново присвояване по стойност, при което се присвояват адреси на обекти, намиращи се в динамичната памет.

Операторът new връща адрес, който може да бъде присвоен на променлива-референция от съответния (или друг подходящ) тип.

В примера, при присвояването „p2=p1”, p2 и p1 сочат към един и същи динамичен обект. Промяната на динамичния обект, чрез която и да е от референциите p1 и p2, се извършва само на едно място.

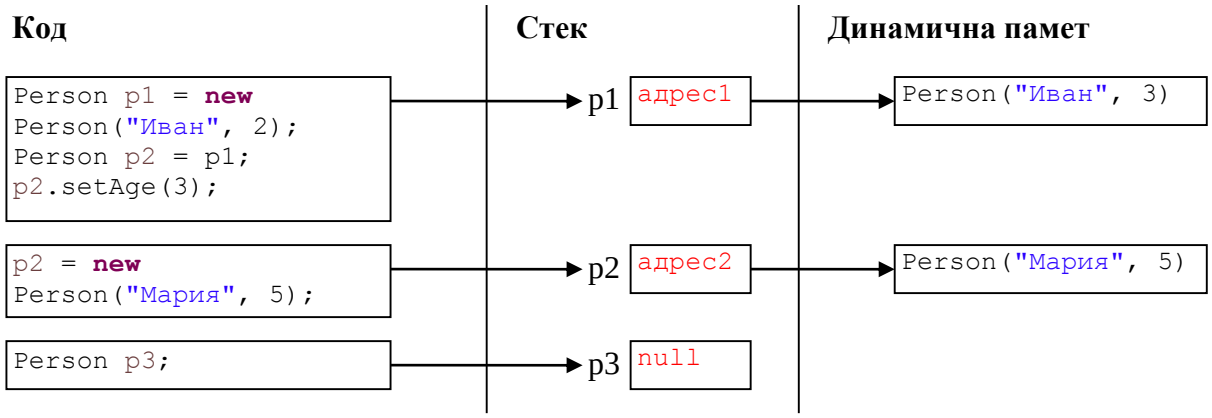


Фиг. 3.3. Присвояване/предаване по адрес

Ако на по-късен етап в кода на програмата се създаде нов динамичен обект и се присвои на една от променливите, то другата променлива ще продължи да сочи към първоначалния обект (фиг. 3.4).

```
// създаване на нов динамичен обект и насочване на p2 към него
p2 = new Person("Мария", 5);
System.out.println(p1);
```

```
System.out.println(p2);

Person p3; // p3 има стойност null
// System.out.println(p3); // получава се грешка при компилиране

Резултат:
Иван е на 3 г.
Мария е на 5 г.
```



Фиг. 3.4. Присвояване/предаване по адрес

Процедурен и обектно-ориентиран подход за реализация на програмите

Java е обектно-ориентиран език за програмиране, но той позволява и процедурен стил на програмиране.

При процедурния подход, в един клас се създават множество статични методи, които се изпълняват по логика, определена в main-метода. Аргументи, влияещи върху изпълнението на методите обикновено се предават чрез параметри.

При обектно-ориентирания подход се създават множество от класове. Обектите на класовете комуникират помежду си чрез методи, а използваните от тях аргументи, освен явните параметри са и полетата на класа.

Анотации

Анотациите – Annotation – са вид интерфейси, които носят мета информация за потребителите и/или за ВМ на Java.

Декларират се като интерфейсите, но имат допълнителен специален символ - @:

```
[<модификатори>] @ <име на интерфейс>
```

Ако в тялото на анотацията не са описани елементи, тя се нарича маркер.

Съществуват няколко стандартни анотации:

- @Override – указва, че методът пренаписва друг метод. Ако не е намерен родителски метод със същата декларация, компилаторът на Java (или използваната среда за разработка) ще изведе съобщение за грешка.
- @Deprecated – указва, че не е препоръчително да се използва методът, защото има по-нова реализация. Множество такива методи се поддържат в стандартните

класове на Java, за да може да се изпълняват програми, написани за по-стари версии на JDK.

- и др.

Например, може да се запише анотация @Override пред описанията на методите s() за изчисляване на лице в класовете Circle и Rectangle, деклариран в родителския интерфейс Surface.

```
public class Cirlce implements Surface{
	public double r;

	@Override
	public double s(){
		return PI*r*r;
	}
}
```

Тази анотация може да се запише и при методите toString(), декларирани първоначално в родителския клас java.lang.Object.

```
	@Override
	public String toString(){
		return name + " е на " + age + " г.";
	}
```

Изключения

Изключенията са специален вид грешки, възникващи по време на изпълнение на програмата, които нарушават изпълнението на желаната последователност от команди в програмата.

Изключения могат да възникнат например, когато вместо число, за вход е въведен низ, който не може да бъде преобразуван до число. Вместо програмата да приключи изпълнението си, изключението може да бъде прихванато и обработено. При това може да се изпълни друг подходящ код, коригиращ грешката или да се изведе подходящо съобщение за възникналата грешка.

Изключенията са вид обекти, представители на клас java.lang.Throwable или на някой от неговите наследници – класовете java.lang.Error и java.lang.Exception. Тези класове имат множество различни наследници, конкретизиращи различни видове грешки.

При възникване на изключение се генерира обект от съответния клас, който се изхвърля (throws) към извикващия метод или се обработва в текущия метод чрез т.нар. try-catch блокове.

В следващия пример е показано описателно начина за обработка на грешки.

```
try{
	// Код, който може да предизвика изключение
}
// Ако възникне изключение от клас Exception (или негов наследник),
// то обекта за изключение се достъпва чрез променливата-референция „e”
catch(Exception e){
	// Код, който се изпълнява при възникване на изключение.
	// Примерен код: извеждаме съобщение и използваме метод
	// getMessage на класа Exception, който връща низ за грешката
	System.out.println("Съобщение за грешка: " + e.getMessage());
}
```

Стил на програмиране

Под стил за програмиране се разбират препоръчителни, но незадължителни правила за задаване на имена, подредба и оформление на кода. Чрез тях се постига прегледност и по-лесна четимост на кода.

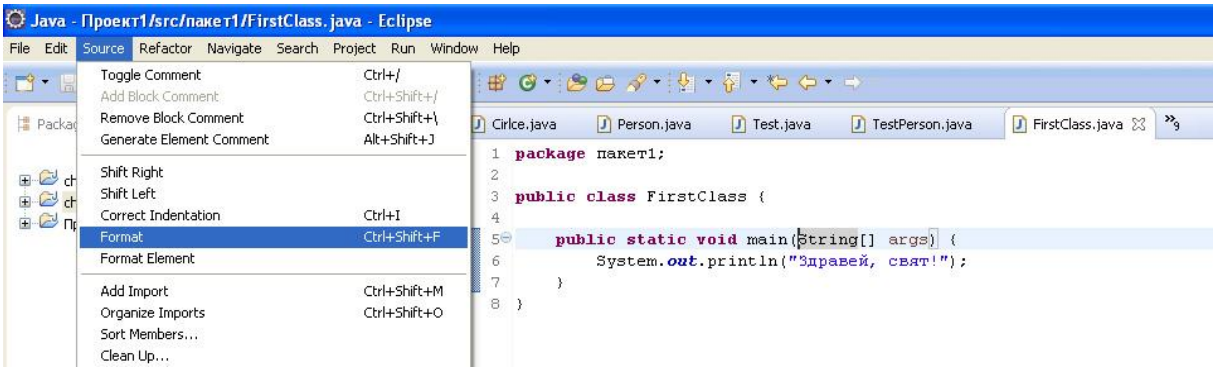
Някои правила (конвенции) за именуване и коментари, приети в Java са:

- имената трябва да са смислени и подсказващи, както следва:
  - класовете и обектите се задават чрез съществителни имена (Document, Cloud, University, Student, Address…);
  - методите описват действия и затова е за предпочитане да се задават като глаголи (toString, getInfo; ако се използват имена на български език, което по-скоро е неудачно, – може да се използват имена от вида отпечатайОценките, сумаНаМасив,,...);
  - интерфейсите добавят характеристики към основния клас и затова имената им често се задават като прилагателни;
- имената може да се състоят от няколко думи, без разделител между тях;
- първите букви на имената на класовете и интерфейсите са главни, а следващите – малки (Person, PersonTest);
- имената на методи, пакети и полета спазват правилата за именуване на класове и интерфейси, но започват с малка буква (person, toString);
- имената на константи се записват само с главни букви (RED, GREEN, BLUE...; JANUARY, FEBRUARY,…);
- при локални променливи от един и същи тип, към имената може да се добавят номера (double d1, d2; Person person1, person2;);
- към всеки метод се задава коментар с описание на логиката, параметрите, връщания резултат;
- части от кода с по-особена (сложна) реализация, се коментират;
- и др.

Има множество правила за форматиране – като места за поставяне на фигурните скоби на блоковете, отмествания и много други, които няма да коментираме, тъй като са илюстрирани чрез предоставяния в примерите код.

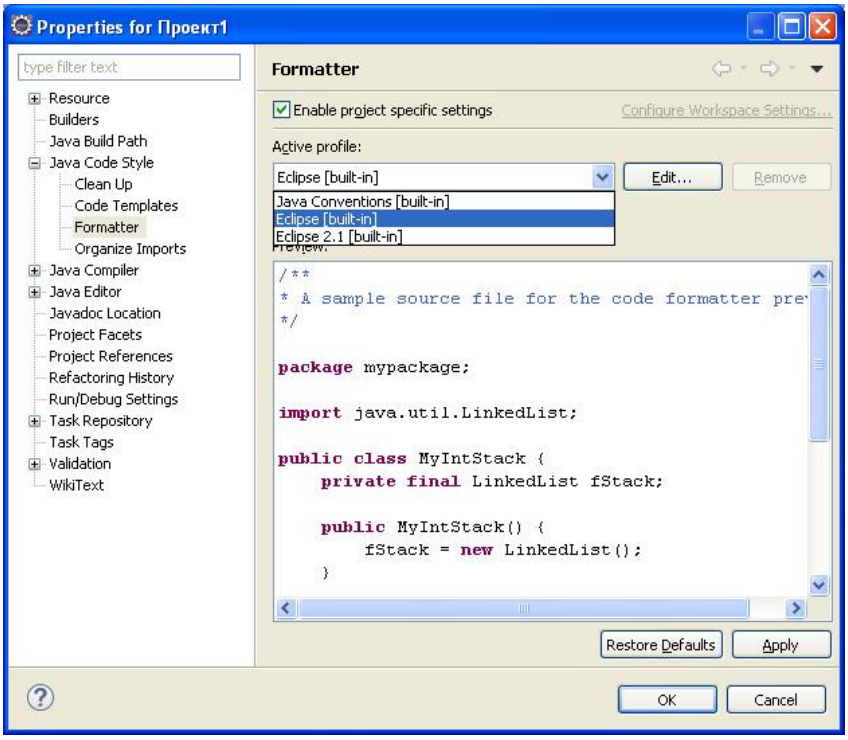
В средите за разработка има вградени правила (инструменти) за автоматизирано форматиране. Те не се прилагат автоматично, ако при писане е допусната синтактична грешка. След поправката на грешките, обаче, то може да се стартира.

На следващия екран е показан начин за автоматично форматиране на текущо отворения файл в Eclipse – от менюто Source/Format.



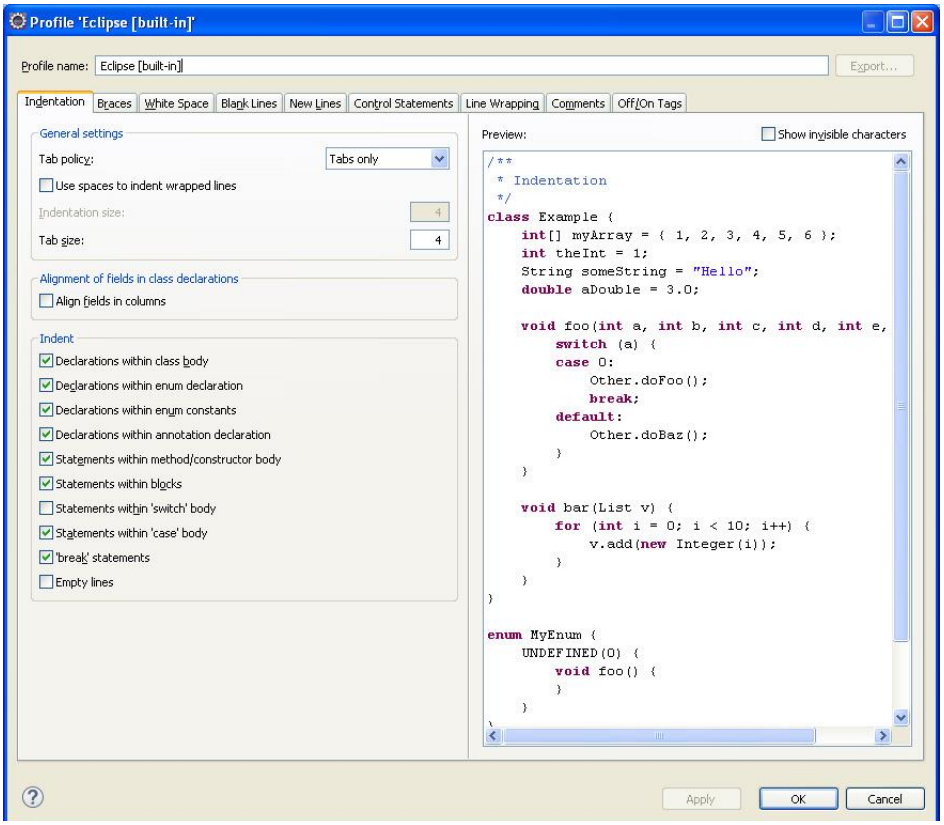
Фиг. 3.5. Автоматично форматиране на кода в Eclipse

В Eclipse може да се избира измежду няколко стандартни множества с правила за форматиране, които да се прилагат автоматично. Опцията е достъпна от менюто Project/Properties, в отворилия се прозорец се избира Formatter/списък Active Profile.



Фиг. 3.6. Избор на стандартно множество от правила за форматиране на кода в Eclipse

От бутон „Edit…” може да се разгледат и редактират стандартните правила за форматиране.



Фиг. 3.7. Редактиране на правила за форматиране на кода в Eclipse

Въпроси и задачи за упражнения

1. Какво определят понятията асоциативност и приоритет при работа с операторите?
2. Какви видовете величини има в Java?
3. Какво представляват литералите?
4. Какво представляват променливите?
5. Каква е разликата между променливи за примитивни и сложни типове?
6. Какво описват методите?
7. Какво видове методите има?
8. Каква е разликата между клас и интерфейс?
9. Какво е конструктор?
10. Как се създава обект от клас?
11. Какво се описва с понятието пакет?
12. Как се осъществява предаването/присвояването по стойност в Java?
13. Как се осъществява предаването/присвояването по адрес в Java?
14. Какви са правилата за именуване в Java?
15. Разгледайте стандартните правила за форматиране на код в Eclipse.

Глава 4. Основни типове и величини. Класове-обвивки на примитивните типове

Всички величини в Java – променливи, константи и литерали, си имат определен тип.

Примитивните типове са свързани с основи понятия като число, символ, булева стойност.

Класът java.lang.Object, е основният сложен тип, а всички останали са негови производни.

Класът String служи за описание и работа с низове.

Той е наследник на Object, но за него са създадени някои улесняващи работата оператори, което го приближава като начин на работа до простите типове.

Освен споменатите типове, в тази глава са разгледани и свързаните с простите типове класове-обвивки.

Представени са и начините за задаване на стойности от прости типове чрез литерали.

Характеристики на основните типове

Всеки тип си има име, размер на заеманата памет, дефиниционна област и стойност по подразбиране.

Тип	**Размер**	**Стойности/Дефиниционна област**	**Стойност по подразбиране**
boolean	1 бит	false и true	false
byte	8 бита	[-128, 127] или $[-2^7, 2^7-1]$	0
short	16 бита	[-32768, 32767] или $[-2^{15}, 2^{15}-1]$	0
char	16 бита	[0, 65535] за числа и от '\u0000' до '\uffff' за символи	0 или '\u0000'
int	32 бита	$[-2^{31}, 2^{31}-1]$	0
long	64 бита	$[-2^{63}, 2^{63}-1]$	0L
float	32 бита	$[-3.4028235*10^{38}, 3.4028235*10^{38}]$	0.0f
double	64 бита	$[-1.7976931348623157*10^{308}$, $1.7976931348623157*10^{308}]$	0.0f
Object	променлив		null

За работа с:

- **цели числа,** се използват типовете **byte, short, int, long**;
- **реални числа с плаваща запетая – float и double;**
- **символи** – **char**;
- **булеви стойности** – **boolean**.

Паметта за съхранение на обект е сума от паметите, заети от полета му. Паметта за референцията към обект зависи от използваната ВМ на Java (32 или 64 бита).

В зависимост от числата, с които ще работи програмата, трябва да се избират подходящи типове. За да не се разхищава памет, ако се работи с малки числа, трябва да се изберат типове с по-малък размер на използваната памет.

Например, за представяне на:

- възраст на човек, може да се използва тип byte (тип за цяло число, с максимална стойност до 127) или short (ако решим, че 127 е недостатъчно);
- оценка, месец – byte (или float за оценка, ако се допускат и не-цели числа);

- година на раждане – short.

Класове-обвивки на примитивните типове

Класовете-обвивки освен алтернативен начин за работа с простите типове предоставят и допълнителни възможности.

В следващата таблица са показани примитивните типове и съответните им класове-обвивки от пакета java.lang.

Примитивен тип	Клас-обвивка
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double
големи цели числа	java.math.BigInteger
реални числа с фиксирана запетая	java.math.BigDecimal

Имената на класовете-обвивки започват с главна буква и са същите като имената на съответните им прости типове с две изключения – Integer (за int) и Character (за char). Има няколко важни класове за работа с числа, които нямат съответни примитивни типове. BigInteger може да се ползва за работа с големи цели числа, ако диапазона на long не е достатъчен, а BigDecimal представя реални числа с фиксирана запетая, при които за разлика от float и double няма загуба при закръгляванията. Действията с BigInteger и BigDecimal се изпълняват по-бавно спрямо примитивните типове.

В следващия пример са показани стойностите на някои от константите, дефинирани в класове-обвивки:

```
public class Wrappers {
    public static void main(String[] args) {
        // Показване на някои константи в класовете-обвивки
        System.out.println("Мин. ст. за byte -> " + Byte.MIN_VALUE);
        System.out.println("Макс. ст. за byte -> " + Byte.MAX_VALUE);

        System.out.println("Мин. ст. за int -> " + Integer.MIN_VALUE);
        System.out.println("Макс. ст. за int -> " + Integer.MAX_VALUE);

        System.out.println("Макс. ст. за double -> " + Double.MAX_VALUE);
        System.out.println("Мин. ненулева положителна ст. за double -> " +
Double.MIN_VALUE);

        System.out.println("Мин. ст. за boolean -> " + Boolean.FALSE);
        System.out.println("Макс. ст. за boolean -> " + Boolean.TRUE);
    }
}

Резултат:
Мин. ст. за byte -> -128
Макс. ст. за byte -> 127
Мин. ст. за int -> -2147483648
Макс. ст. за int -> 2147483647
Макс. ст. за double -> 1.7976931348623157E308
```

```
Мин. ненулва положителна ст. за double -> 4.9E-324
Мин. ст. за boolean -> false
Макс. ст. за boolean -> true
```

Литерали

Литералите са неименувани константи, които се записват в изходния код.

Чрез тях се задават стойности на променливи и именувани константи от примитивните типове и типа String.

Литерали могат да участват и в изрази, но това не е препоръчително, ако литералът се повтаря многократно. Напр., за предпочитане е да създадем константа „final double PI = 3.14;” и да използваме нея, вместо многократно да записваме литерала 3.14. Ако в даден момент решим, че трябва заменим 3.14 с по-точно число 3.1416, в първия случай ще трябва да направим еднократна промяна, а във втория – на повече места, което увеличава възможността за допускане на грешка.

Литералите притежават тип, който се определя по време на компилиране на програмата и който зависи от стойността му.

Примерни литерали са:

- true, false – за тип boolean;
- 1, 25, -25 – за цели числа;
- 3.14, 5.1E2 (=$5.1*10^2$) – double;
- 3.14f – float – ако не се укажат с f, типа на реалните числа се приема за double;
- ‘3’, ‘a’, ‘\n’, ‘\u0410’ – char – символ или ескейп последователност за символ, зададени в апострофи;
- “текст”, „низ” – литералите от тип String се задават като последователност от символи, оградена в кавички.
- null – специална стойност за променливите-референции към обект, която означава че не е зададена стойност.

Правилата за задаване на различните типове литерали са разгледани по-подробно.

Булев тип и литерали

Възможните стойности на булевия тип са литералите true (истина) и false (лъжа). Булеви стойности могат да се получат в резултат от изчислението на логически изрази и да се използват като условия в различни оператори за контрол на изпълнението на програма.

В следващия пример са декларирани булеви променливи b1, b2, b3. За b1 и b2 са зададени стойности литерали, а b3 се изчислява чрез логически израз

```
// Булеви литерали и променливи
boolean b1 = true;
boolean b2 = false;
boolean b3 = (1<2) && (1>2); // логически израз
System.out.println("b1->" + b1);
System.out.println("b2->" + b2);
System.out.println("b3->" + b3);

Резултат:
b1->true
b2->false
b3->false
```

Целочислени типове и литерали

Типовете за цели числа са byte, short, int, long. В описания ред дефиниционните области на типовете са подмножества едно на друг, както следва: byte < short < int < long.

За разлика от други езици за програмиране, в Java примитивните целочислени типовете са само знакови (signed) – описват отрицателни и положителни числа. От версия JDK 8 в класовете-обвивки на примитивните целочислени типове са добавени възможности за работа с беззнакови (unsigned) числа.

Литералите се задават като последователност от цифри, които може да се предхождат от знак + или -.

По подразбиране целочислените литерали получават тип int. Литералите за byte и short се дефинират чрез типа int, а за да се укаже, че литерала е от тип long, след числото се записва буквата ‘L’ или ‘l’.

В следващия пример е показан начини за задаване на литерали за променливи от целочислен тип.

```
byte age = 20;
short year = 2015;
int counter = 33000;
long identificator1 = 5;
long identificator2 = 5L;
```

В Java може да се задават числа в различни бройни системи – с основа 2, 8, 10, 16 Стандартният начин за показване на стойностите е в десетична бройна система, без значение от начина на задаването им.

Особености при представянията на числата в различни бройни системи са:

- двоична – започват с 0b (или 0B); използваните цифри са {0, 1};
- осмична – започват с 0; използваните цифри са {0, 1, 2, 3, 4, 5, 6, 7};
- десетична – използваните цифри са {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
- шестнайсетична – започват с 0x (или 0X); използваните цифри са {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}; цифрите над 9 може да се задават с малки или главни букви.

Тези правила са илюстрирани в следващия примерен код:

```
// Задаване на цели числа в различни бройни системи
int binNumber = -0b110;    // основа 2
int octalNumber = 037;     // основа 8
int decimalNumber = 19;    // основа 10
long hexNumber = -0xFE;    // основа 16

// Стойностите на числата в десетична бройна система
System.out.println("binNumber->" + binNumber);
System.out.println("octalNumber->" + octalNumber);
System.out.println("decimalNumber->" + decimalNumber);
System.out.println("hexNumber->" + hexNumber);

Резултат:
binNumber->-6
octalNumber->31
decimalNumber->19
hexNumber->-254
```

С помощта на методи на класовете-обвивки може да се преобразува число до низ, представящ го в дадена бройна система. За класове Long и Integer съществуват методи toBinaryString, toOctalString, toHexString с параметър число, които преобразуват числото съответно до двоичен, осмичен и шестнайсетичен низ.

```
// Показване на число в различни бройни системи
byte number = 31;
String numberToString = Long.toBinaryString(number);
System.out.println(number + " в двоичен вид -> " + numberToString);
numberToString = Long.toOctalString(number);
System.out.println(number + " в осмичен вид -> " + numberToString);
numberToString = Long.toHexString(number);
System.out.println(number + " в шестнайсетичен вид -> " + numberToString);

Резултат:
31 в двоичен вид -> 11111
31 в осмичен вид -> 37
31 в шестнайсетичен вид -> 1f
```

Клас java.math.BigInteger

При работа с примитивните типове за цели числа трябва да се отчете, че те имат ограничени дефиниционни области. Различни операции могат да доведат до получаване на некоректни стойности поради „препълване" т.е. стойността на дадена променлива да излезе извън обхвата на типа ѝ.

В следващия пример е показано препълване на променлива от тип byte, което може да се случи при всеки друг примитивен тип.

```
byte b = Byte.MAX_VALUE;
System.out.println("b.1 -> " + b); // b.1 -> 127
b++;                               // препълване
System.out.println("b.2 -> " + b); // b.2 -> -128

Резултат:
b.1 -> 127
b.2 -> -128
```

Ако е необходимо да се работи с по-големи от допустимите за типа long числа, може да се използва клас java.math.BigInteger. В обект от този клас се съхранява непроменяемо (immutable) цяло число с произволна дължина. Непроменяемо означава, че при изпълнение на операция (метод) върху обекта, не се променя първоначално съхраненото в него число, а се връща нов обект, представящ резултатното число.

Обект от класа BigInteger се създава с различни конструктори, но най-използван е конструкторът с параметър от тип String. Очаква се числото да е записано коректно, иначе възниква грешка.

Основните операции за работа с числа се извършват чрез методи, които връщат като резултат нов обект от типа BigInteger. Методите се изпълняват за единия обект (ляв операнд), а получават като параметър втория обект (десен операнд):

- събиране – BigInteger add(BigInteger val);
- изваждане – BigInteger subtract(BigInteger val);
- умножение – BigInteger multiply(BigInteger val);
- деление – BigInteger divide(BigInteger val).

Разбира се има много други методи, които може да разгледате самостоятелно.

Следващият код е пример за намиране на сума на две числа от тип BigInteger.

```
import java.math.BigInteger;

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

```
            // Максималната стойност за типа Long като низ
            // Трябва ни, защото за BigInteger няма конструктор
            // с параметър от тип long, а има със String
            String maxLongAsString = "" + Long.MAX_VALUE;

            // Създаване на BigInteger обект за числото maxLongAsString
            BigInteger bIntMaxLong = new BigInteger(maxLongAsString);

            // Създаване на BigInteger обект за числото 1
            BigInteger bIntOne = new BigInteger("1");

            // Метод add се извиква за левия операнд с параметър – десния
            // Записваме резултата в нов обект
            BigInteger bIntSum = bIntMaxLong.add(bIntOne);

            System.out.println("bInt -> " + bIntMaxLong);
            System.out.println("sum -> " + bIntSum);
        }
}

Резултат:
bInt -> 9223372036854775807
sum  -> 9223372036854775808
```

Реални числа с плаваща запетая

Типовете за реални числа с плаваща запетая са float и double. Чрез double (double precision) се представят реални числа в по-голям диапазон спрямо float (single precision).

В компютърните системи не могат да се представят всякакви реални числа. Поради ограничения в стандарта, по който се представят, и определената им памет, реалните числата в десетичен вид съдържат 8/9 (за float) или 16/17 (за double) значещи цифри за цяла и дробна част. По-големи числа се представят, като значещите цифри са умножени със степени на десетката. Например, при float числото 1 000 000 (един милион) се представя в десетичен вид като 1000000.0; десет милиона се представя като $1.0*10^7$;

На ниво битове, реалните числа се описват в двоичен формат, като се използва стандарта IEEE 754. Те заемат 32 или 64 бита (за float и double съответно). Първият бит определя знака на числото – 0 за '+'; 1 за '–'. В следващите битове се записват:

- порядък p (експонента) – цяло число;
- мантиса M– реално число между 1 и 2.

Абсолютната стойност на реалното число е $M*2^p$. В зависимост от порядъка, битовете за мантисата са плаващ брой, откъдето идва и името на числата с плаваща запетая (floating point).

Литерали за реални числа с плаваща запетая

Реалните числа се записват с цифри за цялата и дробната част, а десетичната запетая се означава с '.' (точка).

За да се укаже, че литерал е число от тип float, се записва f (или F) в края, а за double – d (или D). Ако след числото не е указан символ, се приема че то е от тип double.

При задаване на литералите може да се укаже порядък чрез символ E (или e – латински) последван от степента. Например, 1.2E2 е числото $1.2*10^2$, -2.5E-3 – $2.5*10^{-3}$. В следващия пример е показано задаване на стойности на променливи и реално записаните за тях стойности. За удобство на читателя, резултатът от отпечатването в конзолата е показан като коментар след декларациите на променливите.

```
// Литерали и променливи за реални числа с плаваща запетая
```

```
float f1 = 1234567.1f;      // f1 -> 1234567.1
float f2 = 1234567.12345f; // f2 -> 1234567.1
float f3 = 123456789f;      // f3 -> 1.23456792E8
float f4 = 10.1f;           // f4 -> 10.1

System.out.println("f1 -> " + f1);
System.out.println("f2 -> " + f2);
System.out.println("f3 -> " + f3);
System.out.println("f4 -> " + f4);

double d1 = 12345678.12345678;  // d1 -> 1.234567812345678E7
double d2 = 12345678.123456789; // d2 -> 1.234567812345679E7
double d3 = 1.2E2;              // d3 -> 120.0
double d4 = -2.5E-2;            // d4 -> -0.025

System.out.println("d1 -> " + d1);
System.out.println("d2 -> " + d2);
System.out.println("d3 -> " + d3);
System.out.println("d4 -> " + d4);
```

Специални стойности за реални числа с плаваща запетая

Според стандарта IEEE 754, поддържан от Java, към реалните числа с плаваща запетая са добавени 3 специални стойности, равностойни на другите стандартни числа:

- **Infinity** – безкрайност;
- **-Infinity** – минус безкрайност;
- **NaN** (NotANumber) – не число.

В следващата таблица са описани специфични особености при извършване на аритметични оператори върху тези стойности.

Действие	Резултат
число / (+/-) Infinity	0
(+/-) Infinity * (+/-) Infinity	(+/-) Infinity
ненулево число / 0	(+/-) Infinity
Infinity + Infinity	Infinity
(+/-) 0 / (+/-) 0	NaN
Infinity - Infinity	NaN
(+/-) Infinity / (+/-) Infinity	NaN
(+/-) Infinity * 0	NaN

За някои от случаите е показан примерен код на Java:

```
double zero = 0.0;
double one = 1.0;
double infinity = one/zero;

System.out.println(infinity + " -> " + infinity);
System.out.println(zero + "/" + zero + " = " + zero/zero);
System.out.println(one + "/" + zero + " = " + one/zero);
System.out.println(-one + "/" + zero + " = " + (-one/zero));
System.out.println(infinity + "/" + infinity + " = " +
(infinity/infinity));

Резултат:
Infinity -> Infinity
0.0/0.0 = NaN
```

```
1.0/0.0 = Infinity
-1.0/0.0 = -Infinity
Infinity/Infinity = NaN
```

Трябва да се отбележи, че специалните стойности **(+/-)Infinity и NaN се получават само при работа с реални числа от тип float и double**, но не и при цели числа. **Ако при работа с цели числа се извърши деление на 0 се получава грешка**, при което програмата прекъсва изпълнението си:

```
// При работа с цели числа се получава грешка при делене на нула
// и програмата прекъсва изпълнението си
System.out.println("1/0 -> " + 1/0);

Резулата е прекъсване на програмата със съобщение за грешка:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at chapter4.FloatingPointsSpecialValues.main(FloatingPointsSpecialValues.java:8)
```

Реални числа с фиксирана запетая – клас java.math.BigDecimal

Тъй като не всички реални числа с плаваща запетая имат точно представяне, често се получават неочаквани грешки при изчисленията, като е показано в следния код:

```
// Грешки при закръгляване
double PI = 3.14;
double r = 3.0;
System.out.println("PI*r*r -> " + PI*r*r);
System.out.println("r*r*PI -> " + r*r*PI);

Резултат:
3.14*3*3 = 28.259999999999998
3*3*3.14 = 28.26
```

Класът java.math.BigDecimal предоставя възможност за работа с реални числа с фиксирана запетая. Обектите от този клас са непроменяеми (immutable). Представят се с две цели числа:

- за цялата част – има произволна дължина;
- за дробната част – 32-битово число.

Броят на символите в дробната част се нарича мащаб (scale).

Подобно на класът BigInteger, BigDecimal най-сигурно се инициализира чрез низ – запис на реално число. При инициализация чрез реално число от тип double или float, може да се получи некоректна стойност за BigDecimal.

Също както и при BigInteger, за основните аритметични действия с обекти на класа BigDecimal се използват методи add(), subtract(), multiply(), divide().

Въпреки че с BigDecimal може да се записват числа с много повече значещи цифри, отколкото при примитивните типове float и double, основното предимство е в това, че при числата с фиксирана запетая няма загуба на точност. BigDecimal се използва най-вече в програми, в които се работи с валути и други мерни единици, при които дробната част не е много голяма (напр. 2 символа), но загубата на точност е нежелана.

В следващия код е реализирана формулата за лице от предходния пример, като е използван типа BigDecimal.

```
import java.math.BigDecimal;
```

```
public class FixedPoint {
      public static void main(String[] args) {
            BigDecimal bPI = new BigDecimal("3.14");
            BigDecimal bR = new BigDecimal("3");
            BigDecimal s = bPI.multiply(bR).multiply(bR);
            System.out.println("s -> " + s); // 28.26
      }
}

Резултат:
s -> 28.26
```

Съвместимост на типове

Два типа са съвместими, ако обхватът на единия е подмножество или съвпада с обхвата на другия.

При числовите типове може да се каже че byte<short/char<int<long<float<double. Типът char обикновено се използва за символи, но може да представя и числа (от 0 до 65535). Типовете short и char са частично съвместими.

При обектните типове (наследници на клас Object) има съвместимост между Object и всички останали класове. Не всички класове са съвместими помежду си, въпреки че са в обща йерархия в Java. Само класовете от една линия на наследяване, имащи преки или косвени наследствени връзки помежду си са съвместими. Това е така, защото наследниците притежават всички полета на родителите и съответно, възможните им стойности са разширено множество на възможните стойности на родителя.

За два съвместима типа X и Y може да се каже, че X е **под-тип**, а Y **над-тип**, ако възможните стойности на X са подмножество на възможните стойности на Y.

Преобразуване по тип

Преобразуването по тип е понятие, свързано със съвместимостта между типове.

Може да се преобразува величина от един тип в друг, ако двата типа са съвместими.

Преобразуването от под-тип към над-тип (напр, byte в int) не води до загуба на точност и се извършва неявно, без да е изрично указано.

Ако се преобразува величина от над-тип към под-тип, може да се получи загуба на точност, ако стойността е извън обхвата от множеството от възможни стойности на под-типа. Преобразуването в такъв случай може да се извърши само със специален **оператор за преобразуване по тип**

(<тип>) <величина> ,

чрез който се указва преобразуване на стойност от някакъв тип към зададения в скобите тип.

В следващия пример са показани възможни, но не винаги коректни преобразувания на променливи от примитивни типове.

```
byte b = 43;
int i = b;    // неявно преобразуване по тип
float f = i; // неявно преобразуване по тип

// i = f;     // не може неявно да се преобразува
i = (int) f; // но може да се преобразува явно от float към int
```

```
// Следват примери за преобразуване със загуба на точност
i = (int) 354.9; // Преобразуване на double до int,
                 // със закръгляване надолу
b = (byte) i;    // Преобразуване на int до byte

System.out.println("i -> " + i);
System.out.println("b -> " + b);

Резултат:
i -> 354
b -> 98
```

Литерали за типове char и String

Символният тип char представя символите от кодовата таблица UNICODE.

Символите се използват и като елементи на текстовите величини от тип String.

Символ може да се представи по различни начини като литерал, заграден в апострофи:

- единичен символ – 'a', 'b';
- с код на символ – '\u0061', '\u0062' – кода се задава с ляво наклонена черта '\', последвана от u и четири шестнайсетични цифри;
- ескейп (escape) последователност– '\специален символ' – предоставят алтернативен запис на специални символи, които не винаги могат да се запишат като обикновен символ.

Има няколко относително често използвани ескейп последователности в Java, описани в следващата таблица:

ескейп последователност	символ
\t	табулация
\n	преминаване на нов ред
\'	апостроф
\"	кавичка
\\	ляво наклонена черта

Тъй като апострофите, кавичките и ляво наклонената черта са специални символи, в определени случаи е необходимо да се ескейпнат.

В следващия примерен код е показана работа със символи и низове. Демонстриран е начин за преобразуване на char до int и обратно, което дава възможност за манипулиране на символите. За именуване на променливите са използвани български букви.

```
public class Chars {
  public static void main(String[] args) {
    char буквaA = 'A';     // българска буква А
    char буквaБ = 'Б';     // българска буква Б
    char апостроф = '\'';
    char кавичка1 = '\"'; // всъщност, кавичка като символ...
    char кавичка2 = '"';  // ... може да не се ескейпва

    System.out.println("\u0430\u0431\u0432"); // абв

    // В низ кавичките трябва се ескейпват
    System.out.println("ул. \"Цар Асен\" № 24");

    System.out.println("\nКодове на български букви:"); //2 нови реда
    System.out.println(буквaA + " -> " + (int)буквaA);
```

```
    System.out.println(буквaБ + " -> " + (int)буквaБ);

    int кодНаБуквaБ = (int)буквaБ;
    int кодНаБуквaВ = кодНаБуквaБ + 1;
    char буквaВ = (char)кодНаБуквaВ;
    System.out.println(буквaВ + " -> " + (int)буквaВ);
    // Накратко за буква Г
    char буквaГ = (char)((int)буквaВ + 1);
    System.out.println(буквaГ + " -> " + (int)буквaГ);
  }
}
```

Резултат:

```
абв
ул. "Цар Асен" № 24

Кодове на български букви:
А -> 1040
Б -> 1041
В -> 1042
Г -> 1043
```

Преобразуване от низ в число

Често входните данните на програмите се получават във вид на низове, но е необходимо да бъдат обработвани като числа или булеви стойности.

За да се преобразува низ до стойност от определен примитивен тип се използват специални методи за парсиране (parse – в превод синтактичен/ морфологичен разбор) в класовете-обвивки. Методите са статични и затова обикновено се използват чрез името на класа. Имената им са Boolean.parseBoolean(String), Byte.parseByte(String), Short.parseShort(String), Integer.parseInt(String), Long.parseLong(String), Float.parseFloat(String), Double.parseDouble(String).

В следващия код са показани начини за парсиране на низ до различни примитивни типове:

```
String s = "123";
// int i = (int)s; // не може така да се преобразува String до int
int i = Integer.parseInt(s);  // парсиране на String s до int
float f = Float.parseFloat(s);

double d = Double.parseDouble("-45e5");

System.out.println("i -> " + i); // i -> 123
System.out.println("f -> " + f); // f -> 123.0
System.out.println("d -> " + d); // d -> -4500000.0

// Само "true" се парсира като true
boolean b = Boolean.parseBoolean("true");
System.out.println("b.1 -> " + b);  // b.1 -> true

// други низове се парсират до false
b = Boolean.parseBoolean("5.123");
System.out.println("b.2 -> " + b);  // b.2 -> false
```

При използване на методите за парсиране може да възникнат изключения. За коректна работа на приложенията е необходимо тези изключения да бъдат прихващани и обработвани, както е показано в следващия код. Низ, получен като параметър е

парсиран до цяло число чрез метод parseToInt(String), след което резултатът се отпечатва в конзолата. При възникване на грешка при парсирането се извежда съобщение. В main метода може да се извиква многократно създаденият метод parseToInt.

```
// Методът преобразува параметъра s до цяло число и извежда информация
public static void parseToInt(String s){
	int i=0; // в i ще запишем парсираната стойност на низа s

	try{
		i = Integer.parseInt(s); // опитваме да парсираме
	}
	catch(Exception e){ // при грешка се влиза в catch-блока
		System.out.println("Низът '" + s + "' не е цяло число.");
	}

	// Методът продължава да се изпълнява дори да е възникнала грешка
	System.out.println("i -> " + i);
}

public static void main(String[] args) {
	String s = "123.4"; // получаваме s отнякъде
	parseToInt(s);		// извикваме нашия метод parseToInt с параметър s

	s = "123";
	parseToInt(s);		// второ извикване на метода
}

Резултат:
Низът '123.4' не цяло число.
i -> 0
i -> 123
```

Въпроси и задачи за упражнения

1. Какви са основните характеристики на типовете?
2. Какви са стойностите на основните характеристики за примитивните типове?
3. Потърсете в Интернет начин за намиране на текущия размер на обект.
4. Напишете програма, в която се демонстрират всички начини за получаване на специалните стойности (+/-)Infinity и NaN, при работа с реални числа с плаваща запетая.
5. Разгледайте самостоятелно и тествайте методите на класовете BigDecimal и BigInteger.
6. Напишете метод, който получава като параметър две цели числа от тип long и създава BigDecimal обект, за който цялата част е първото число, а дробната – второто. Върху създадения обект и друг произволен обект покажете основните действия за работа с числа. Използвайте метода с различни примерни стойности за параметрите.
7. Предложете поне десет физически характеристики, описващи човек, студент, работник, фирма или някакъв друг обект (изберете един обект). Задайте подходящи имена за променливи и типове за тях.

Упътвания към задачите

4. Използвайте таблицата с особени случаи, представена в частта Специални стойности за реални числа с плаваща запетая.
6. За да създадете параметър-низ за конструктора на BigDecimal, конкатенирайте long-параметрите с низ ".", помежду им.

Глава 5. Оператори и изрази

Операторите извършват **действия върху** различни типове **данни**, наричани **операнди или аргументи.** В резултат от прилагането на оператор се получава стойност от тип, който може да не съвпада с типа на участващите данни.

Основните видове оператори са аритметични, логически, побитови, за сравнение, за присвояване, конкатенация на символни низове. Те са разгледани по-подробно в тази глава.

Израз е комбинация от оператори, операнди и скоби за указване на приоритета им, която спазва определени синтактични правила.

Правилата, по които се изчисляват сложните изрази, се определят от приоритета и асоциативността на участващите оператори.

Приоритетът определя реда на изпълнение на операторите. Ако не сме сигурни какъв е приоритетът на операторите в даден израз, може да използваме скоби.

Асоциативността определя в какъв ред операндите се присъединяват към оператора. Има ляво-асоциативни оператори (със стандартна логика), при които първо се изчислява изразът от ляво на оператора, а след това десният; и дясно-асоциативни, при които операторът се прилага едва след изчислението на израза от дясната му страна.

Оператор за съединяване на символни низове

Операторът + за конкатенация на низове (от тип String) към края на левия операнд залепва десния операнд и връща като резултат получения низ. Задължително е единият операнд да е низ, а другият може да е от произволен тип. При съединяването, операндът автоматично се преобразува до низ. Ако операндът е обект, автоматично се извиква метода му toString(), който е дефиниран в класа Object и може да бъде пренаписан във всеки негов наследник.

Знакът + се интерпретира като знак за конкатениране на низове, ако поне единият операнд е низ, и като знак за сумиране на числа, ако двата операнда са числа. Операторът за конкатениране на низове има по-голям приоритет от оператора за сумиране. Това означава, че ако има сложен израз от вида „низ+число1+число2” първо ще се извърши слепването на низ с число1 и ще се получи нов низ, а след това ще се изпълни вторият оператор +, който ще съедини получения низ с число2. За да се изпълни вторият знак + като сума на числа, е необходимо да се използват скоби. Тези особености са показани в следващия код:

```
// тук и двата знака + са за съединяване на низове...
String infoSum1 = "Сумата на числата 5 и 6 е " + 5 + 6;

System.out.println(infoSum1);  // Сумата на числата 5 и 6 е 56

// ... а тук (5 + 6) се изпълнява с приоритет като сумиране
String infoSum2 = "Сумата на числата 5 и 6 е " + (5 + 6);
System.out.println(infoSum2);  // Сумата на числата 5 и 6 е 11
```

Оператор за присвояване

Основният оператор за присвояване е = . Логиката му е, че на променлива от лявата му страна се присвоява стойността на израза от дясната му страна. Типа на израза в дясно трябва да е съвместим с типа на променливата в ляво.

```
int i = 5;
i = 6+3;
```

Операторът за присвояване е дясно-асоциативен, което означава че първо се пресмята израза в дясно, а след това резултатът се присвоява на променливата в ляво.

Има и други оператори за присвояване, които се комбинират с аритметични и побитови оператори.

Аритметични оператори

Операндите на аритметичните оператори са числа, резултатът също е число.

Логиката на повечето аритметични оператори ни е известна от математиката:

Оператор	Име	Примери
+	събиране	a+b
-	изваждане	a-b
+	унарен плюс	+a
-	унарен минус	-a
*	умножение	a/b
/	деление	5/1→5, 5/2→2, 5/3→1, 5/4→1, 5/5→1
%	остатък при целочислено деление	5%1→0, 5%2→1, 5%3→2, 5%4→1, 5%5→0 5.2%2.1→1.0, 5.2%1.0→0.2
i++ ++i	унарно увеличаване с единица (increment)	int i= 2; i++; i→3; ++i; i→4;
i-- --i	унарно намаляване с единица (decrement)	int i= 2; i--; i→1; --i; i→0;

Унарните оператори ++ и -- може да се задават както преди, така и след променлива, чиято стойност трябва да се промени с единици:

- когато променливата е в дясно от оператора (++i, --i), той е дясно-асоциативен;
- когато променливата е в ляво от оператора (i++, i--), той е ляво -асоциативен.

Подобно на унарния минус -i, дясно-асоциативните варианти ++i и --i се изпълняват с най-голям приоритет при участието им в изрази. За разлика от тях ляво-асоциативните (i++ и i--) се изпълняват след като израза се изчисли. Това е показано в следващия примерен код:

```
public class IncrementOperator {
      public static void main(String[] args) {
            int k=10;

            // ляво-асоциативен ++;
            // първо k се предава към метода за отпечатване
            // след това k се увеличава с 1
            System.out.println(k++);      // 10

            // дясно-асоциативен ++; първо k се увеличава с 1
            // след това k се предава към метода за отпечатване
            System.out.println(++k);      // 12

            // тук асоциативността не е от значение
            k++;
            ++k;
            System.out.println(k);        // 14
      }
   }
```

Има някои особености в типа на резултата, при използване на аритметичните бинарни оператори:

- При операнди от целочислени типове byte, short и int, резултата винаги е от тип int. Не може, например, за три променливи “byte b1, b2, b3;” да запишем “b3= b1+b2;”. Може да запишем “int i=b1+b2”.
- При използване на операнд от тип long и друго цяло число, резултата е long.
- Ако единият операнд е от целочислен, а другия от реален тип, резултата е реално число.
- Важно е да се отбележи, че при деление на цели числа, резултатът е цяло число. Ако е необходимо при деление на две цели числа да се получи реално, трябва да се преобразува поне единият операнд до тип за реално число.

В следващия пример са създадени два статични метода, в които са показани бинарните аритметични оператори върху цели и реални числа. Методите имат еднакви имена и по два параметъра. Различават се само по типовете на параметрите. В main-метода може многократно да се извикват тестващите методи с различни стойности на параметрите.

```
public class ArithmeticOperators {
	public static void test(int i1, int i2){
		System.out.println(i1 + "+" + i2 + "=" + (i1 + i2));
		System.out.println(i1 + "-" + i2 + "=" + (i1 - i2));
		System.out.println(i1 + "*" + i2 + "=" + (i1 * i2));
		System.out.println(i1 + "/" + i2 + "=" + (i1 / i2));
		System.out.println(i1 + "%" + i2 + "=" + (i1 % i2));
	}

	public static void test(double d1, double d2){
		System.out.println(d1 + "+" + d2 + "=" + (d1 + d2));
		System.out.println(d1 + "-" + d2 + "=" + (d1 - d2));
		System.out.println(d1 + "*" + d2 + "=" + (d1 * d2));
		System.out.println(d1 + "/" + d2 + "=" + (d1 / d2));
		System.out.println(d1 + "%" + d2 + "=" + (d1 % d2));
	}

	public static void main(String[] args) {
		test(5, 3);
		test(7, 3);
		test(7.0, 3);
	}
}

Резултат:
5+3=8, 5-3=2, 5*3=15, 5/3=1, 5%3=2
7+3=10, 7-3=4, 7*3=21, 7/3=2, 7%3=1
7.0+3.0=10.0, 7.0-3.0=4.0, 7.0*3.0=21.0, 7.0/3.0=2.3333333333333335, 7.0%3.0=1.0
```

В примера са използвани два различни оператора +: единият е за събиране на числа, а другият – за конкатениране на низове. Операторът за конкатениране е с по-голям приоритет. За да се промени приоритета и за да се изпълнят коректно операциите събиране и изваждане на числа, са използвани скоби: (i1 + i2), (i1 - i2).

Оператори за сравнение

Операндите на операторите за сравнение са числа, а резултатът е булева стойност.

Оператор	Име	Примери с литерали	
==	равно	1==2→false	2==2→true
!=	различно	1!=2→true	2!=2→false
<	по-малко	1<2→true	2<2→false
>	по-голямо	1>2→false	2>2→false
<=	по-малко или равно	1<=2→true	2<=2→true
>=	по-голямо или равно	1>=2→false	2>=2→true

Операторите == и != се използват и за сравнение на променливи-референции със стойността null. Ако референцията има стойност null, това означава, че тя не сочи към съществуващ обект в динамичната памет. Резултатът от сравнението на две референции е истина, ако те сочат към един и същи обект, и е лъжа, ако реферират различни обекти. В следващия пример е използван клас за човек с едно поле age.

```
// Примерен клас за човек
class Person{
      int age=1;
      // ... и други полета и методи
}
// Клас за сравнение на референции с оператор ==
public class ReferencesEquality {
      public static void main(String args[]){
        Person p1 = new Person();
        Person p2 = p1;  // p1 и p2 реферират един и същи динамичен обект
        Person p3 = new Person(); // p3 реферира друг динамичен обект

        // булеви променливи за резултати от сравнения
        boolean equals_P1_P2 = (p1==p2);  // скобите не са задължителни
        boolean equals_P1_P3 = (p1==p3);
        System.out.println("equals_P1_P2 -> " + equals_P1_P2);
        System.out.println("equals_P1_P3 -> " + equals_P1_P3);
        System.out.println("p1==null -> " + (p1==null));
        System.out.println("p1!=null -> " + (p1!=null));
      }
}

Резултат:
equals_P1_P2 -> true
equals_P1_P3 -> false
p1==null -> false
p1!=null -> true
```

Въпреки че данните в p1 и p3 са равни (age има стойност 1) сравнението на обектите p1==p3 връща false, защото те реферират различни динамични обекти.

Ако е необходимо да се сравнява дали данните в два обекта имат еднакви стойности, трябва да се използва друг подход – например, да се пренапише метода equals, наследен от класа Object. За множество различни стандартни класове, за които сравнението има смисъл, този метод е пренаписан.

Изводът, който може да направим засега е, че трябва да се внимава с използването на операторите == и != при референции.

Логически оператори

Операндите на логическите оператори са булеви стойности, а резултатът също е булева стойност.

Оператор	Име
&&	Логическо И
||	Логическо ИЛИ
!	Отрицание - НЕ
^	Изключващо или

В глава 2 са разгледани булевата алгебра и правилата, по които могат да се създават и опростяват логически изрази. Тук само са припомнени таблиците за истинност, които показват какъв е резултата при всевъзможните комбинации от стойности на операндите и са дадени примери за някои от операторите.

a	**b**	**a && b**
false	false	false
false	true	false
true	false	false
true	true	true

a	**b**	**a || b**
false	false	false
false	true	true
true	false	true
true	true	true

a	**b**	**a ^ b**
false	false	false
false	true	true
true	false	true
true	true	false

a	**!a**
false	true
true	false

Пример за логическо И:

До участие в конкурс се допускат лица, които са навършили 16 години и представят автобиография (CV).

За да участва конкретен човек (Иван) в конкурса трябва да удовлетворява едновременно и двете условия.

Пример за логическо ИЛИ:

За участие в състезание се допускат лица, които са навършили 12 години или са с придружител.

Достатъчно е само едно от условията да е изпълнено, за да се допусне Иван до конкурса:

- да е по-голям от 12 г.;
- да е с придружител;
- може да са изпълнени и двете условия едновременно – по-голям от 12 г. и с придружител.

Пример за логическо отрицание:

За участие в конкурс се допускат лица, които не са по-малки от 12 г.

Отричаме (не допускаме) участието на хора по-малки от 12 г.

Понякога може да се избегне отрицание: В примера, може да се изисква лицата да са по-големи от 12 г. т.е. ***!(<12г.) е равносилно на (>12г.)***.

Условен оператор ?:

Създадените чрез логическите оператори условия се използват в различни конструкции за контрол на изпълнението на кода (които са разгледани в глава 7), а също и в условния оператор.

Чрез тройния условен оператор ?: се изчислява един от два възможни израза, в зависимост от истинността на някакво логическо условие.

Синтаксисът на тройния оператор е следният:

```
<логическо условие> ? <израз_true> : <израз_false>;
```

Първият операнд е логическо условие. Другите два операнда са изрази, чийто резултатен тип обикновено е един и същ (но това не е задължително). Ако логическото условие има стойност true, оператора връща като резултат стойността на израза след ?, иначе – израза след :.

Когато резултатът от изпълнението на условния оператор се присвоява на променлива, двата израза трябва са от тип, съвместим с типа на променливата.

Работата с условния оператор е показана в следващия пример. Създаден е метод, който получава като параметри име и възраст на участник, и булева стойност, указваща дали лицето има или няма придружител. Условието за допускане на участник в конкурса е: възраст над 12 години или лицето да е с придружител. В резултат от изпълнението, методът извежда в конзолата информация за това дали даден участник се допуска в конкурс. В основната програма се проверяват няколко човека.

```
public class Competition {

      // Метод: извежда информация може ли човек да участва в конкурс
      // name - име, age - възраст, companion - има ли придружител
      public static void testPerson(String name, int age, boolean
companion){
            boolean test = ( (age>12 || companion) ? true : false );

            String info = name + (test?"":" не")
                  + " може да участва в конкурса. Възраст: "
                  + age + ". Придружител: " + (companion?"да":"не") + ".";

            System.out.println(info);
      }
      public static void main(String[] args) {
            testPerson("Иван", 12, false);
            testPerson("Мария", 12, true);
            testPerson("Петър", 13, false);
      }
}

Резултат:
Иван не може да участва в конкурса. Възраст: 12. Придружител: не.
Мария може да участва в конкурса. Възраст: 12. Придружител: да.
Петър може да участва в конкурса. Възраст: 13. Придружител: не.
```

В примера данните за участниците са зададени като литерали. За да се въведат нови участници, трябва да се промени програмата. Това, разбира се, не е много добре. Програмите трябва да се правят така, че да не се променят често. В следващата глава са представени различни начини за въвеждане на данни.

Многократно влагане на условния оператор

Единият от операндите на условния оператор може да е друг условен оператор. Ако например знаем, че цяло число приема стойности 1, 2, 3 и желаем да отпечатаме неговата стойност като текст, може да напишем следния израз:

```
   String s = ( i==1 ? "едно": (i==2?"две":"три") );
```

Този запис е малко объркващ. Има и по-подходящи начини за обработка на няколко условия – условни конструкции и конструкция за избор от варианти. Те са разгледани в глава 7.

Побитови оператори

Побитовите оператори се изпълняват върху битовото (двоичното) представяне на числа. Операндите са числа, резултата също е число.

Основните побитови (bitwise) оператори са &, |, ~, ^. Другите са битово-изместващи (bitshift) – <<, >>, >>>.

В глава 2 е разгледана логиката на някои побитови оператори в частта Операции върху двоични числа. Всички побитови оператори в Java са представени в следващата таблица. Двоичните числата са получени чрез използването на метода Integer.toBinaryString(int), който не показва водещите нули.

Оператор	Име	Примери с числа-литерали	
		десетични	**двоични**
&	И	7&4 -> 4 13&10 -> 8	111&100 -> 100 1101&1010 -> 1000
|	ИЛИ	7|4 -> 7 13|10 -> 15	111|100 -> 111 1101|1010 -> 1111
~	НЕ	~3 -> -4 ~-3 -> 2	~11 -> 11111111111111111111111111111100 ~11111111111111111111111111111101 -> 10
^	Изключващо ИЛИ	7^4 -> 3 13^10 -> 7	111^100 -> 11 1101^1010 -> 111
<<	Изместване в ляво	32<<1 -> 64 31<<1 -> 62	100000 << 1 -> 1000000 11111 << 1 -> 111110
>>	Изместване в дясно без знака	32>>1 -> 16 31>>1 -> 15 5>>1 -> 2 -5>>1 -> -3	100000 >> 1 -> 10000 11111 >> 1 -> 1111 101 >> 1 -> 10 11111111111111111111111111111011 >> 1 -> 11111111111111111111111111111101
>>>	Изместване в дясно със знака	5>>>1 -> 2 -5>>>1 -> 2147483645	101 >>> 1 -> 10 11111111111111111111111111111011 >>> 1 -> 11111111111111111111111111111101

Операторът i<<n измества числото i с n бита на ляво и може да се използва за умножение по 2^n. Операторите i>>n и i>>>n изместват числото i с n бита на дясно и може да се използват за деление на 2^n със закръгляване надолу. Разликата между последните два оператора се вижда при отрицателните числа: При >> в изместването не участва водещият бит за знак на числото, а при >>> – участва.

Код за работа с побитово И е показан в следващия пример.

```
public class BitOperators {

      // Метод, демонстриращ побитово И
      public static void bitwiseAND(int i1, int i2){
            int result = i1&i2;
            String s1 = Integer.toBinaryString(i1);
            String s2 = Integer.toBinaryString(i2);
            String binary_result = Integer.toBinaryString(result);

            System.out.print(i1 + " & " + i2 + " = " + result);
            System.out.println("  ->  " + s1 + " & " + s2 + " = " +
binary_result);
      }
```

```
      public static void main(String[] args) {
            bitwiseAND(7, 4);
            bitwiseAND(15, 10);
            bitwiseAND(13, 10);
      }
}

Резултат:
7 & 4 = 4   ->  111 & 100 = 100
15 & 10 = 10   ->  1111 & 1010 = 1010
13 & 10 = 8   ->  1101 & 1010 = 1000
```

Още оператори за присвояване

Освен стандартния оператор за присвояване, има и други оператори, свързани с бинарните аритметични и побитови оператори.

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

Левият операнд е променлива, в която се записва резултата от изпълнението на аритметичния/побитов оператор заедно с десния операнд.

Действието на сложните оператори за присвояване може да се изрази чрез стандартни оператори. Напр.,

```
i += 3;  е равносилно на  i = i + 3;
i %= 3;  е равносилно на  i = i%3;
```

Други оператори

За пълнота на изложението, в следващата таблица са описани и останалите основни оператори.

Оператор	Име	Описание
(тип)	Преобразуване по тип	Преобразува операнд до указан тип. Може да се използва само за съвместими типове.
instanceof	Проверка за инстанция	Изисква два операнда - десният е референция към обект, а левият – име на клас или интерфейс. Резултатът е true, ако левият операнд е инстанция на десния, в противен случай е false.
new	Създаване на обект	Операндът след new е конструктор на клас. Резултатът е референция към създаден динамичен обект.
.	Достъп до член на обект, клас, интерфейс	Левият операнд е име на клас/интерфейс или референция към обект. В дясно се записва член (поле, метод,...) на класа.
(параметри)	Обръщение към метод	В скобите се задават фактически параметри, а преди тях - име на метод. Резултатът може да е от всякакъв тип.
[]	Достъп до елементите на масив	Пример: array[i]. Преди скобите се записва име на масив, а в скобите се задава индекс на елемент на масива. Резултатът е от типа на елементите на масива.

Изрази

Комбинацията от няколко оператора и операнда се нарича израз.

Има прости изрази, състоящи се от един оператор или дори само операнд (литерал, променлива, константа или функция, която връща стойност) и сложни изрази, съдържащи повече от един оператор.

Възможно е в един израз да участват операнди от различни типове. Как точно се изчислява един сложен израз и какъв е крайният му тип, се определя от приоритета и асоциативността на участващите оператори.

Приоритет и асоциативност на операторите

В следващата таблица са показани всички оператори, като в горната част са тези с най-висок приоритет, а надолу приоритетът им намалява. Операторите в един ред от таблицата са с еднакъв приоритет. Може да се използват скоби за указване или промяна на приоритета на операторите. Скобите правят изразите по-прегледни и намаляват вероятността от допускане на случайни логически грешки, когато те (изразите) са сложни.

Оператор	Име	Асоциативност
[] . () i++ i--	достъп до елементите на масив достъп до членове извикване на метод i++ увеличаване с единица i-- намаляване с единица	от ляво на дясно
++i --i + - ! ~	++i увеличаване с единица --i намаляване с единица унарен плюс унарен плюс логическо отрицание побитово отрицание	от дясно на ляво
(тип) new	преобразуване по тип създаване на обект	от дясно на ляво
* / %	умножение деление остатък при целочислено деление	от ляво на дясно
+	съединяване на низове	от ляво на дясно
+ -	сума разлика	от ляво на дясно
<< >> >>>	побитово изместване наляво побитово изместване надясно (без знака) побитово изместване надясно (със знака)	от ляво на дясно
< <= > >= instanceof	по-малко по-малко или равно по-голямо по-голямо или равно инстанция ли е	от ляво на дясно
== !=	равно различно	от ляво на дясно
&	побитово И	от ляво на дясно
^ ^	побитово Изключващо ИЛИ логическо Изключващо ИЛИ	от ляво на дясно
|	побитово ИЛИ	от ляво на дясно
&&	логическо И	от ляво на дясно
||	логическо ИЛИ	от ляво на дясно
?:	условен оператор	от дясно на ляво
= оператор=	присвояване комбинирани оператори за присвояване	от дясно на ляво

Асоциативността определя реда на присъединяване на операндите към оператора.

Повечето оператори са ляво-асоциативни. При тях първо се изчислява левият операнд, а след това – десният, и накрая се изпълнява операторът.

При дясно-асоциативните, първо се изчислява десният операнд, след него левият, и накрая се прилага оператора. В следващия пример, за израза „j = 3 + -i;” имаме унарен минус, който е дясно-асоциативен, ляво-асоциативен плюс и дясно-асоциативен оператор за присвояване. Унарният минус е с най-голям приоритет и ще се изпълни първи върху неговия десен операнд i. След това ще се изпълни събирането на числото 3 с новополученото – 4 – и крайният резултат ще се присвои на j.

```
int i=-4, j;
j = 3 + -i;
```

Въпроси и задачи за упражнения

1. Създайте и използвайте метод, получаващ две реални числа чрез параметри и демонстриращ върху тях действието на операторите за сравнение.
2. Какъв е резултатът от изпълнението на командите `int i = 5; i = i++; System.out.println(i);`? Защо?
3. Създайте методи, показващи действията на побитовите оператори.
4. Напишете метод, показващ дали число, зададено като параметър, е четно или нечетно.
5. Създайте и използвайте метод за намиране лице на триъгълник по зададени основа и височина.
6. Напишете и използвайте метод, при който по зададени разстояние и средна скорост на движение се извежда времето, необходимо за изминаване на разстоянието.
7. Създайте и използвайте метод за извеждане на разстоянието от точка с координати (x,y) до центъра на двумерна декартова координатна система.
8. Създайте и използвайте метод за намиране на разстоянието между две точки (x1, y1) и (x2, y2) в двумерна декартова координатна система.
9. Напишете програма за намиране на разстоянието между 2 точки в тримерна координатна система.
10. Нека с флагове-числа, които са степени на 2, означим различни неща:
    - f1=1 – има храна;
    - f2=2 – има вода;
    - f3=4 – има ток;
    - f4=8 – има Интернет;
    - и т.н (допълнете сами).

    В една променлива flags, като сума от флагове е записано какво има даден човек. Например 0-означава, че няма нищо; 7 (1+2+4) – има храна, вода и ток, но няма Интернет и т.н. Създайте и използвайте метод, който получава като параметри името и числото flags за човека, и извежда информация какво има този човек.
11. Напишете и използвайте метод/и, при който/които по зададено число n се извеждат химичните формули за алкани, алкени, алкини с n въглеродни атома.

Упътвания към задачите

1. Вижте примера за аритметичните оператори.
2. Вижте приоритетите и асоциативността на операторите = и i++.

3. Разгледайте метода за побитово И в класа BitOperators. Създайте подобни методи за другите оператори.
4. Ако остатъка при деление на 2 е 0, то числото е четно.
5. s=(a*h)/2.
6. t = s/v.
7. Разстоянието се пресмята по питагоровата теорема. Използвайте Функцията Math.sqrt(число) за коренуване.
8. Методът получава 4 параметъра за координатите на 2-те точки. Подобно на задача 5, катетите са |x1-x2| и |y1-y2|.
9. Помислете – формулата е подобна на формулата в двумерна координатна система.
10. Използвайте оператора побитово И, за да проверите за всеки отделен флаг дали се съдържа във flags.
11. Използвайте общите формули на съединенията: алкани – C_nH_{2n+2}, алкени – C_nH_{2n}, алкини – C_nH_{2n-2}.

Глава 6. Задаване на входни данни. Помощни класове

В примерите от предните глави, входните данни бяха твърдо кодирани в кода на програмата. За да се изпълни програма с различни входни данни, трябва да се промени кодът и да се компилира. Програмата може да получава входни данни преди стартиране или по време на работа (в run-time), без да е необходимо да се компилира наново. Това може да стане по различни начини – чрез задаване на параметри при стартиране, въвеждане от потребителя през конзолата или чрез графичен интерфейс, четене от файл или от база данни и др.

В тази глава са разгледани различни начини за задаване на входни данни, като за целта са представени и някои използвани за целта класове.

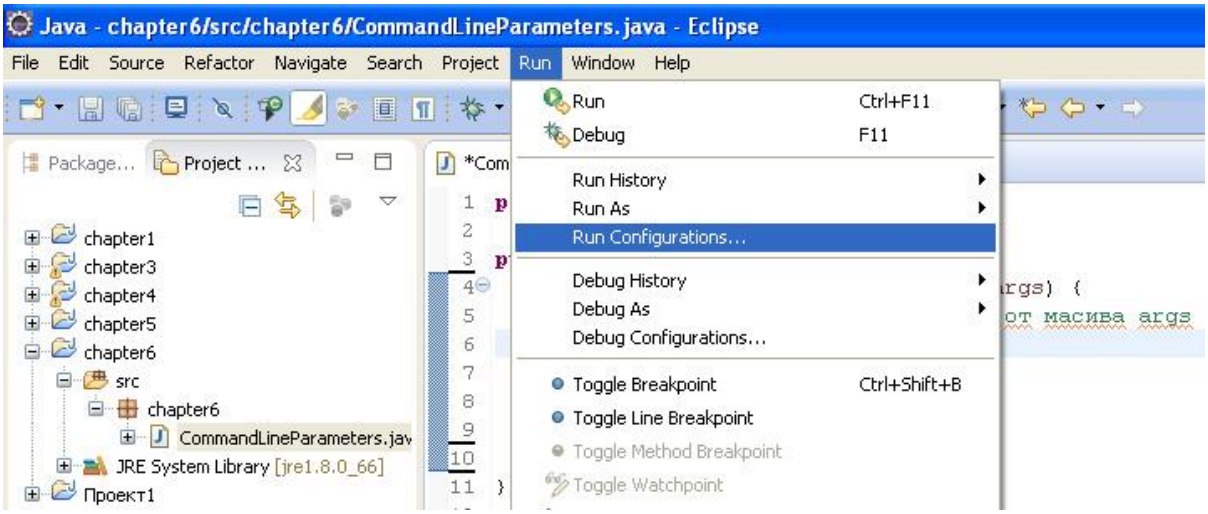
Параметри от командния ред

При стартиране на програма в конзолата, може да се задават входни аргументи след името на класа:

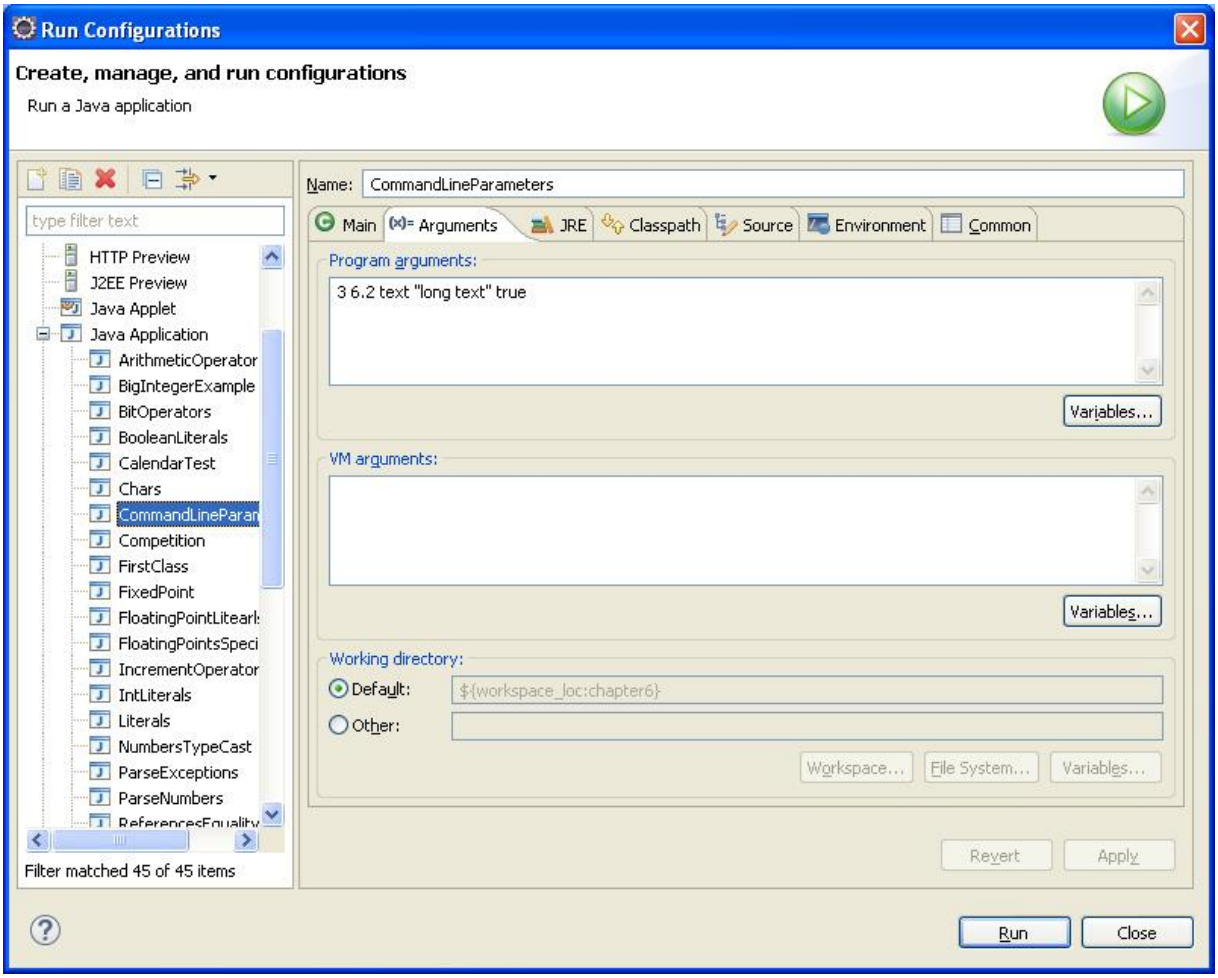
```
java <class-файл> [<аргументи разделени със интервал>]
```

Аргументите се предават към main-метода на програмата като параметър – масив с елементи от тип String.

В Eclipse задаването на тези параметри става от менюто „Run/Run Configurations...” (фиг. 6.1). Отваря се прозорец (фиг. 6.2), в чийто таб „Arguments” трябва да се избере клас от списъка с класове, и в текстовото поле „Program arguments” да се запишат параметрите, разделени с интервал. Ако някой от параметрите е низ, съдържащ интервали, низът трябва да бъде ограден в кавички.



Фиг. 6.1. Екран от Eclipse – меню „задаване на параметри от командния ред”



Фиг. 6.2. Екран от Eclipse – задаване на параметри от командния ред

В следващия примерен код е показано как могат да бъдат прочетени входните параметри:

```
public class CommandLineParameters {
      public static void main(String[] args) {
            // Четене на входни параметри от масива args
            System.out.println(args[0]);
            System.out.println(args[1]);
            System.out.println(args[2]);
            System.out.println(args[3]);
            System.out.println(args[4]);
      }
}

Резултат:
3
6.2
text
long text
true
```

Обръщението към несъществуващ елемент на масив може да предизвика грешка. Тъй като не може да се разчита, че потребителите винаги ще въвеждат коректен брой входни данни, е добре да се използва try-catch блок за обработване на изключенията. Друга

особеност е, че входните параметри винаги се получават като низове. За да се използват като числа или булеви стойности, трябва да се преобразуват с методите за парсиране от съответните класове-обвивки. Тези методи също могат да предизвикат възникването на грешки, ако потребителят не е задал параметри от очаквания в програмата тип. В следващия пример се използват аргументите за намиране на лице или периметър на правоъгълник. Първите два аргумента са числа, петият – указва дали търсим лице (ако е true) или периметър (ако е false), а останалите параметри не се използват.

```
public class CommandLineParameters {
       /*
        * Програма за намиране на лице или периметър на правоъгълник
        * Очаквани входни параметри:
        * args[0] - страна a - цяло число
        * args[1] - страна b - реално число
        * args[2], args[3] - описания, които няма да ползваме
        * args[4] - лице ли да се намери (при true) или периметър (при false)
        */
  public static void main(String[] args) {
    try{
       int a = Integer.parseInt(args[0]);
       double b = Double.parseDouble(args[1]);
       boolean calculateSquare  = Boolean.parseBoolean(args[4]);

       double rezult = calculateSquare ? a*b : 2*(a+b);

       System.out.println((calculateSquare?"Лицето":"Периметъра")
             + " на правоъгълник със страни " + a + " и " + b + " е " + rezult);
    }catch(Exception e){
       System.out.println("Моля, въвеждайте коректни входни параметри - " +
             "int double String String boolean!");
    }
  }
}

Резултат при различни аргументи:
3 6.2 text "long text" true:
Лицето на правоъгълник със страни 3 и 6.2 е 18.6
3 6.2 text "long text" false:
Периметъра на правоъгълник със страни 3 и 6.2 е 18.4
3.1 6.2 text "long text" false:
Моля, въвеждайте коректни входни параметри - int double String String boolean!
```

При обработката на грешки може да се прихване точната причина за възникването на грешката и програмата да извежда по-точни съобщения. Засега, за да не се усложни логиката на примерите, try-catch блокове не се използват.

Вход и изход към конзолата

Конзолата е приложение, чрез което потребителите може да комуникират с програмата по време на нейната работа. Всяка операционна система притежава конзола, а в езиците за програмиране са създадени механизми за комуникация с нея. Средата Eclipse също предоставя конзола.

В Java механизмите за вход и изход към конзолата са достъпни чрез класа System. Съществуват два изходни обекта – потоци, с имена Sysetm.out и System.err, които извеждат данни в конзолата. Чрез Sysetm.out се извеждат съобщения с методите print() и println(). По аналогичен начин може да се изведат и съобщения за грешки в System.err (с методите print), но той обикновено се използва за автоматичното извеждане на текста на неприхванатите в catch-блок изключения. Тъй като извеждането на данни в изходните потоци се осъществява паралелно, при възникване на грешки отпечатвания текст в

стандартния изходен поток System.out (получен преди възникване на грешката) се „преплита” с текста за грешката.

Обект за вход от конзолата е System.in. Съществуват различни начини за обработка на постъпващите входни данни, но от версия 1.5 на Java е създаден удобен начин за четене с помощта на класа java.util.Scanner. Чрез обект от класа Scanner може да се чете не само входния поток от конзолата – System.in – но и други потоци, като низове и файлове.

Локализация – клас java.util.Locale

Особеност при въвеждане на реалните числа от конзолата е, че се използват локализирани настройки за представянето им, т.е. в зависимост от регионалните и езиковите настройки на операционната система, може да се очаква въвеждането на реални числа в различни формати. Например, ако настройките са за България, ще се очаква десетичната запетая да е ‘,’ (запетая), а не стандартната за Java ‘.’ (точка).

Класът java.util.Locale съхранява характеристики за локализирано представяне на различни неща, според специфични за потребителя държава, език и др. –: формати за дата, валута, посока на писане (от ляво-надясно или от дясно-наляво, отгоре-надолу или обратно) и др.

Локализацията представя географски, политически или културен регион. За представянето ѝ се използват различни стандартни характеристики:

- ISO 639 двубуквен или трибуквен езиков код;
- ISO 15924 четирибуквен код за скрипт – напр. Latn за Latin, Cyrl за Cyrillic;
- ISO 3166 двубуквен код на държавата или UN M.49 трицифрен код на област.;
- варианти и допълнения.

Класът Locale използва множество ресурсни файлове, в които са описани характеристиките на различните локализации. За някои по-често използвани локализации са създадени константи, напр. Locale.US, Locale.UK, Locale.ENGLISH. Други трябва да бъдат създадени изрично – като обекти на класа. Например, за България може да се създаде обект за локализация, като в конструктора се укаже ISO 639 двубуквен код на българския език и ISO 3166 двубуквен код на държавата:

```
Locale locale1 = new Locale("bg");        // език
Locale locale2 = new Locale("bg", "BG"); // език, държава
```

Клас java.util.Scanner за четене от входен поток

Създаването на обект Scanner за четене от конзолата става чрез конструктор, на който се задава обекта System.in.

```
Scanner scanner = new Scanner(System.in);
```

След това се използват методи next…() за четене на различни типове данни, които може да се записват в променливи. По време на изпълнение на програмата, срещането на метод next…() прекъсва работата ѝ, изчаква въвеждането на текст и натискане на клавиш Enter на клавиатурата. При това, въведеният низ се прочита, преобразува се до избрания тип (в зависимост от next-метода) и програмата продължава изпълнението си.

В следващия пример е използван метод nextInt(), за прочитането на две цели числа и извеждане на сумата им. Разбира се, при въвеждане на грешни данни, както и при използването на Integer.parseInt() ще се получи грешка.

```
import java.util.Scanner;

public class ConsoleParameters {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);

		System.out.print("Въведете i = ");
		int i = scanner.nextInt();
		// next-метода изчаква въвеждане на текст и натискане на Enter

		System.out.print("Въведете j = ");
		int j = scanner.nextInt();

		scanner.close();

		System.out.println(i + "+" + j + "=" + (i + j));
	}
}

Примерен резултат (зависи какво сме въвели):
Въведете i = 4
Въведете j = 5
4+5=9
```

За да се въведе текст от конзолата, първо трябва да отидем в прозореца на конзолата (с щракане на мишката).

Затварянето на обекта за сканиране с метода close() прекъсва връзката му с входния поток. След като обектът за сканиране е затворен, при опит за използването му ще се получи грешка.

В класа Scanner има различни методи за четене:

- nextShort(), nextByte(), nextInt(), nextLong(), nextDouble(), nextFloat(), nextBoolean() – четат стойности от съответния примитивен тип, описан в името на метода;
- nextBigInteger, nextInteger – четат големи числа от съответните класове;
- next() – за четене на низ (String) без разделители (интервал, таб);
- nextLine() – за четене на низ с включени разделители.

Като параметър за next-методите, връщащи числови типове, може да се задава основа на бройна система за въвеждане. Напр., scanner.nextInt(2) ще очаква въвеждане на число в двоична бройна система.

Методът nextLine чете всички символи до символите за край на ред. Ако преди това е използван друг метод за четене (напр., nextInt()) и е натиснат клавиш Enter, за метода nextLine ще останат само символите за край на ред (от натискането на Enter) и работата му ще приключи. В такъв случай може да се използва nextLine два пъти, като първият е само за да се прескочат символите за край на ред или метода skip, за който може да се укажат символи, които да се прескочат – а именно "\r\n".

```
	scanner.skip("\r\n"); // или scanner.nextLine(); за скипване на Enter
	System.out.print("Въведете s = ");
	String s = scanner.nextLine();
```

В обобщение: **Прескачане на символите за край е необходимо само ако преди nextLine е използван друг next-метод.**

Понякога е необходимо реалните числа да се въвеждат със запетая, а не с точка, както е стандартното им представяне в Java. За да се смени текущата локализация за обект от класа Scanner, се използва методът useLocale(), на който като параметър се задава обект

от клас java.util.Locale. Например, в английска локализация се използва точка за десетична запетая:

```
// за да въвеждаме реалните числа с десетична точка
scanner.useLocale(Locale.ENGLISH);
double d = scanner.nextDouble();
```

В следващия пример е показано използването на методите nextDouble и nextLine.

```
import java.util.Scanner;
import java.util.Locale;

public class ConsoleRealAndStringParameters {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);

		System.out.print("Въведете реално число с точка за десетична
запетая d = ");
		// за да въвеждаме реалните числа с десетична точка
		scanner.useLocale(Locale.ENGLISH);
		double d = scanner.nextDouble();

		System.out.print("Въведете низ без интервали s1 = ");
		String s1 = scanner.next();

		scanner.skip("\r\n"); // или scanner.nextLine(); за скипване
		System.out.print("Въведете низ със интервали s2 = ");
		String s2 = scanner.nextLine();

		scanner.close();

		System.out.println("Реално число: " + d);
		System.out.println("Низ s1: " + s1);
		System.out.println("Низ s2: " + s2);
	}
}

Резултат(в зелено са оцветени примерни въведени от потребителя данни):
Въведете реално число с точка за десетична запетая d = 3.5
Въведете низ без интервали s1 = низ1
Въведете низ със интервали s2 = текст с интервали
Реално число: 3.5
Низ s1: низ1
Низ s2: текст с интервали
```

Възможно е едновременно въвеждане на няколко параметъра, разделени с интервал. Всеки изпълнен next-метод ще обработва частта от низа до следващия интервал.

```
import java.util.Scanner;

public class ConsoleSomeParametersInOneLine {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);

		System.out.print("Въведете на един ред две цели числа и низ: i
j s1 = ");
		int i, j;
		String s;
```

```
            i = scanner.nextInt();
            j = scanner.nextInt();
            s = scanner.next();

            System.out.println(i + "+" + j + "=" + (i + j));
            System.out.println(s);

            scanner.close();
      }
}
```

Примерен резултат:

```
Въведете на един ред две цели числа и низ: i j s1 = 21 4 низ
21+4=25
низ
```

Освен System.in, като параметър на конструктор на Scanner може да се зададе и низ, който да бъде обработван по разгледания в последния пример начин. Низът може да е записан директно, да е получен от командния ред или да е съдържанието на текстов файл. Напр.,

```
    Scanner scanner = new Scanner("4 6.4 true");
```

Използване на графични компоненти за вход и изход

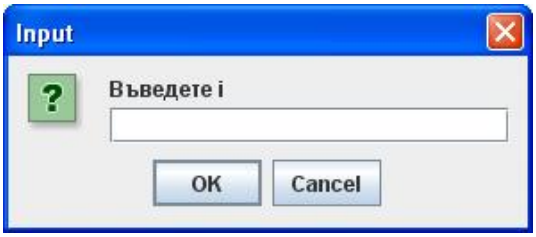
Графичните (GUI – Graphical User Interface – графичен потребителски интерфейс) приложения не са предмет на разглеждане на настоящата книга. Тук само е показано как може да се използват няколко стандартни класа за въвеждане и извеждане на информация чрез графични компоненти.

Графичните компоненти са организирани в пакета javax.swing. Първо трябва да се създаде обект от класа javax.swing.JFrame чрез конструктор:

```
    JFrame frame = new JFrame();
```

Този фрейм се използва като параметър на статичния метод showInputDialog на класа JOptionPane, чрез който се създава графична форма с едно входно поле (фиг. 6.3). Вторият параметър на метода е текст, който подсеща потребителя какво да въвежда. Резултатът от изпълнението на метода се записва в променлива от тип низ. Напр.,

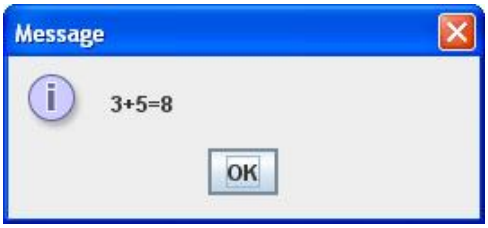
```
    String tempString = JOptionPane.showInputDialog(frame, "Въведете i");
```



Фиг. 6.3. Стартиране на метода JOptionPane.showInputDialog(frame, "Въведете i");

След натискане на някой от бутоните „OK” или „Cancel”, се визуализира форма, която съдържа върнатия от метода низ.

В аналогичен прозорец (фиг. 6.4), получен чрез метода showMessageDialog на класа JoptionPane, се извежда съобщение-резултат. Параметри за метода са вече създаденият фрейм и низът за съобщението.



Фиг. 6.4. Стартиране на метода JOptionPane.showMessageDialog (frame, "3+5=8");

Пълен пример, в който чрез използването на графични компоненти се въвеждат две цели числа и се извежда тяхната сума,, е показан в следващия код.

Ще отбележим отново, че ако входният низ по някаква причина не може да се преобразува до цяло число (например е натиснат клавиш „Cancel”) или не е въведено цяло число, се получава изключение.

```
import javax.swing.*;

public class JOptionPaneInputDialog {
  public static void main(String[] args) {
    // Създаване на фрейм
    JFrame frame = new JFrame();

    // въвеждане на низ от графичен компонент
    String tempString = JOptionPane.showInputDialog(frame, "Въведете i");

    // преобразуване на входния низ до int
    int i = Integer.parseInt(tempString);

    tempString = JOptionPane.showInputDialog(frame, "Въведете j");

    int j = Integer.parseInt(tempString);

    // Подготвяне на низ за резултата
    String message = i + "+" + j + "=" + (i + j);

    // извеждане в конзолата
    System.out.println(message);

    // извеждане в графичен прозорец
    JOptionPane.showMessageDialog(frame, message);
  }
}
```

Генериране на случайни числа - клас java.util.Random

Понякога, при тестване на създадените методи, не е необходимо да се получават точни входни данни. В такива случаи може да се използва стандартният клас java.util.Random за генериране на случайни реални и цели числа от типовете float, double, int, long, както и булеви стойности. Реалните числа се генерират в интервали (0, 1) или (-2, 2), а целите – в дефиниционните области на съответния тип, като за int може да се укаже и интервал от 0 до зададено като параметър число. Използват се методи next() за създаден обект от класа Random, което е демонстрирано в следващия пример.

```
import java.util.Random;

public class RandomValues {
  public static void main(String[] args) {
    // Създаване на обект от клас Random
    Random r = new Random();

    // Извикване на методи за създадения обект
    int intValue1 = r.nextInt();             // в ДО на int
    int intValue2 = r.nextInt(100);          // int в интервала [0, x)
    long longValue = r.nextLong();           // в ДО на long
    float floatValue = r.nextFloat();        // float  между 0.0 и 1.0
    double doubleValue1 = r.nextDouble();    // double между 0.0 и 1.0
    double doubleValue2 = r.nextGaussian(); // double  между -2.0 и 2.0
    boolean booleanValue= r.nextBoolean();  // boolean стойност

    System.out.println("intValue1 -> " + intValue1);
    System.out.println("intValue2 -> " + intValue2);
    System.out.println("longValue -> " + longValue);
    System.out.println("floatValue -> " + floatValue);
    System.out.println("doubleValue1 -> " + doubleValue1);
    System.out.println("doubleValue2 -> " + doubleValue2);
    System.out.println("booleanValue -> " + booleanValue);
  }
}

Случаен резултат:
intValue2 -> 52
longValue -> 1357260431283118670
floatValue -> 0.6991969
doubleValue1 -> 0.09567046815906344
doubleValue2 -> -1.2351576545279008
booleanValue -> false
```

От версия 8 на Java, класът Random предоставя и методи за генериране на потоци от случайни числа в определени интервали. Методите doubles(), ints(), longs() връщат обекти-потоци от класове от пакета java.util.stream: DoubleStream за double, IntStream за int и LongStream за long числа. В методите е добре да се указват параметри за брой на генерираните числа, защото може да възникне грешка „липса на памет”. Чрез потоците пък може да се получат масиви, които са обяснени по-подробно в глава 8. Тук, без допълнителни обяснения, ще покажем само примерен начин за работа с представените класове и показване на генерираните случайни стойности.

```
import java.util.Arrays;
import java.util.Random;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class RandomStreams {
      public static void main(String[] args) {
            Random r = new Random();

            // генериране на масив с 3 double числа в интервал от 0 до 10
            DoubleStream ds = r.doubles(3, 0, 10);
            System.out.println(Arrays.toString(ds.toArray()));
```

```
            // генериране на масив с 10 int числа в интервал от -10 до 20
            IntStream is = r.ints(10, -10, 20);
            System.out.println(Arrays.toString(is.toArray()));

            // генериране на масив с 3 long числа в ДО на long
            LongStream ls = r.longs(3);
            System.out.println(Arrays.toString(ls.toArray()));
      }
}

Случаен резултат:
[6.943472378386437, 0.09170898723555587, 4.507989180565807]
[1, 4, 11, 8, 2, 13, 9, 1, -5, -6]
[-9207675393601598655, 1473266916893800855, -1668173661421831019]
```

Стандартни математически константи и функции – java.lang.Math и java.lang.StrictMath

В класа java.lang.Math са реализирани множество математически функции, повечето от които връщат като резултат реално число от тип double: логаритмични, експоненциални, тригонометрични, обратни тригонометрични, хиперболични, за закръгляване и др. Представени са и двете най-често използвани в математиката константи – числото π и неперовото число ‘e’ – основа на натуралния логаритъм. Някои от по-често използваните функции са показани в следващия демонстрационен пример, с кратки коментари за действието им. В отделни методи сме групирали различните видове математически методи. Използваме създадени от нас методи за отпечатване println, с цел спестяване на по-дългите многократни обръщения към метода System.out.println.

```
public class MathDemonstartion {
  // Метод за отпечатване на реални числа
  public static void println(double d) {
    System.out.println(d);
  }

  // Метод за отпечатване на цели числа
  public static void println(int i) {
    System.out.println(i);
  }

  // Метод за отпечатване на константите от класа Math
  public static void constants() {
    println(Math.PI); // пи - 3.141592653589793
    println(Math.E);  // неперово число - 2.718281828459045
  }

  // Основни числови функции
  public static void mainFuncs() {
    println(Math.abs(-6));    // 6 - абсолютна стойност
    println(Math.log(Math.E));// 1.0 - логаритъм от число зададено като параметър
при основа Math.E
    println(Math.exp(1));     // 2.718281828459045 - степен 1-ва (в примера) на
неперовото число
    println(Math.sqrt(4));    // 2.0 - корен квадратен от параметъра
    println(Math.pow(4, 2));  // 16.0 - повдига първия параметър на степен втория
    println(Math.pow(27, (double)1/3)); // 3.0 - и коренува (корен трети от 27)
    println(Math.min(3, 5));  // 3 - минималното от две числа
    println(Math.max(3, 5));  // 5 - максималното от две числа
  }

  // Закръгляване на числа
  public static void roundFuncs() {
    println(Math.ceil(4.3));  // 5.0 - закръгляване към по-голямото цяло число
```

```
    println(Math.floor(4.9)); // 4.0 - закръгляване към по-малкото цяло число
    println(Math.round(4.6)); // 5.0 - закръгляване към по-близкото цяло число
    println(Math.round(4.4)); // 4.0
    println(Math.round(4.5)); // 5.0
  }

  // Тригонометрични функции
  public static void trigonometricFuncs() {
    println(Math.sin(Math.PI/2)); // 1.0 - sin от ъгъл, зададен в радиани
    println(Math.cos(Math.PI/2)); // трябва да е 0 - cos от ъгъл, зададен в радиани
    // реално се показва 6.123233995736766E-17
    // поради стандартните грешки при работа с double числа

    println(Math.tan(Math.PI/2)); // безкрайност-тангенс от ъгъл, зададен в радиани
    println(Math.sin(Math.PI/2)/Math.cos(Math.PI/2)); // == tan(Math.PI/2)

    println(Math.cos(Math.PI/2)/Math.sin(Math.PI/2)); // 0 - за котангенс няма
специална функция
  }

  // Радиани и градуси - 2*PI радиана = 360 градуса
  public static void radiansAndDegrees(){
    println(Math.PI/2);                    // 1.5707963267948966
    println(Math.toRadians(90));           // 1.5707963267948966
    println(Math.sin(Math.toRadians(90))); // 1.0
    println(Math.toDegrees(Math.PI/2));    // 90.0
  }

  public static void main(String[] args) {
    constants();          // константи
    mainFuncs();          // основни функции
    roundFuncs();         // функции за закръгляване
    trigonometricFuncs(); // тригонометрични функции
    radiansAndDegrees();  // преобразуване на радиани в градуси и обратно
  }
}
```

Класът java.lang.StrictMath реализира същите функции като Math, но част от използваните алгоритми са по-точни. Докато при Math, върху едни и същи входни данни, може да се получат различни резултати за различните платформи, то при StrictMath резултатите са винаги едни и същи.

Въпроси и задачи за упражнения

1. Създайте програма, която получава като параметри от командния ред едно цяло и пет реални числа (a, b, c, d, e). В зависимост от стойността на първия параметър намира сумата (1), разликата (2) или произведението им (3), или изчислява функцията a^b добавя минималното от двете числа c и d и стойността на ‘e’, закръглена към по-близкото цяло число (4).
2. Реализирайте програмата от задача 1 като въвеждате параметрите от конзолата.
3. Реализирайте програмата от задача 1 чрез графичен вход и изход.
4. Реализирайте програмата от задача 1 като генерирате случайни числа.

Упътвания към задачите

1. Използвайте вложени условни оператори и примера от частта за Параметри от командния ред.
2. Използвайте примерите от частта за Клас java.util.Scanner за четене от входен поток.
3. Използвайте примера в частта Използване на графични компоненти за вход и изход.
4. Използвайте примера в частта Генериране на случайни числа - клас java.util.Random.

Глава 7. Конструкции за контрол на изпълнението

Конструкциите за контрол предоставят начини за многократно изпълнение на група от оператори, както и изпълнението на група оператори при определени условия.

В тази главата са разгледани различните варианти на основните видове конструкции за контрол на изпълнението.

Условна конструкция if

Чрез конструкцията if се указва **част от код**, който **се изпълнява само ако е изпълнено определено условие**. Синтаксисът на конструкцията if е следният:

```
if(<условие>){
   [<тяло>]
}
```

Условието е логически израз. Тялото може да съдържа различни команди – изрази, конструкции и др. То се изпълнява само ако стойността на условието е true.

Ако тялото се състои само от една команда, фигурните скоби може да се пропуснат.

Като демонстрация на конструкцията if, в следващия пример е създаден метод, който получава като параметър число и извежда абсолютната му стойност. За целта, ако стойността на параметъра е по-малка от нула, се умножава по -1.

```
public class IfStatement {

	public static void abs(int i){
		int absValue = i;

		if(i<0){
			absValue = -i;
		}

		System.out.println("Абсолютната стойност на " + i
				+ " е " + absValue);
	}

	public static void main(String[] args) {
		abs(5);
		abs(-6);
	}
}

Резултат:
Абсолютната стойност на 5 е 5
Абсолютната стойност на -6 е 6
```

Условна конструкция if-else

При конструкция if-else, в зависимост от това, дали дадено условие е изпълнено, се изпълнява само единият от два блока с оператори. Синтаксисът на конструкцията if-else е:

```
if(<условие>){
   [<тяло-if>]
}else{
   [<тяло-else>]
}
```

Ако условието има стойност true се изпълнява първият блок (тяло-if), иначе се изпълнява блокът, описан след else (тяло-else).

Използването на if- else конструкция прави метода abs по-лесен за четене:

```
public static void abs(int i){
  int absValue;

  if(i<0){
      absValue = -i;
  }else{
      absValue = i;
  }

  System.out.println("Абсолютната стойност на " + i + " е " + absValue);
}
```

За такива елементарни примери, разбира се, може да се използва и условния оператор ?:, който изчислява един от два възможни израза. С условните конструкции обаче, в тялото им може да се изпълняват множество команди.

Друг стандартен пример за използване на конструкцията if-else е намиране на максималното от две числа, а логиката му е подобна на тази за намирането на абсолютна стойност:

```
public class MaxFind {
      // Метод за намиране на максималното от числата d1 и d2
      public static void max(double d1, double d2){
            double max;       // в max ще запомним максималното число

            if(d1>d2){        // ако d1>d2 значи...
                  max = d1;   // ... d1 е максималното,
            }else{            // иначе...
                  max = d2;   // ... d2 е максималното
            }

            System.out.println("Максималното от числата " + d1
                                    + " и " + d2 + " е " + max);
      }

      public static void main(String[] args) {
            max(3, 4);
            max(-5, -8);
      }
}
```

Резултат:

```
Максималното от числата 3.0 и 4.0 е 4.0
Максималното от числата -5.0 и -8.0 е -5.0
```

Вложени условни конструкции

В множество задачи логиката на решението изисква използването на повече условия. Например, ако за две цели числа i1 и i2 трябва да се изведе информация дали първото е по-малко, равно или по-голямо от второто, трябва да се направят поне две сравнения. Ако първото сравнение i1<i2 е истина, ще се изведе съобщението „i1 е по-малко от i2”, иначе за да се определи кой от останалите два варианта е истина, трябва да извърши още едно сравнение – напр., i1==i2, което е необходимо да се вложи в else-блока на първото. Такова решение е показано в следващия пример:

```
public class NumbersComparison {
      public static void comparisonInfo(int i1, int i2){
            String info;

            if(i1<i2){
                  info = i1 + " е по-малко от " + i2;
            }else{
                  if(i1==i2){
                        info = i1 + " е равно на " + i2;
                  }else{
                        info = i1 + " е по-голямо от " + i2;
                  }
            }
            System.out.println(info);
      }

      public static void main(String[] args) {
            comparisonInfo(3, 5);
            comparisonInfo(6, 6);
            comparisonInfo(8, 1);
      }
}

Резултат:
3 е по-малко от 5
6 е равно на 6
8 е по-голямо от 1
```

Конструкция за избор от варианти switch-case

При конструкцията switch-case (switch-превключвам/преминавам към case-случай), **в зависимост от конкретната стойност на израз се изпълнява една от множество групи от команди.** Синтаксисът на конструкцията switch-case е следният:

```
   switch(<израз>){
       case <стойност1>: [<команди1> [break;]]
       case <стойност2>: [<команди2> [break;]]
   ...
       case <стойностN>: [<командиN> [break;]]
       [default: [<команди по подразбиране>]]
   }
```

Типът на израза може да бъде някой от примитивните byte, short, char, int (също и техните класове-обвивки), String или изброим ENUM тип. След всяка ключова дума case се задава стойност-литерал от типа на израза, а след символа ‘:’ се задават съответни команди. Списъкът от стойности съдържа поне един елемент. Ако при изпълнение на програмата, стойността на израза съвпадне с някоя от описаните case-стойности, започват да се изпълняват съответните на case-случая команди. Частта по подразбиране се означава с default и не е задължителна. Тя се изпълнява само ако стойността на израза не съвпадне с никоя от изброените case-стойности.

Командата за прекъсване break не е задължителна и се използва, за да се прекъсне изпълнението на следващите групи от команди. т.е. **Ако не се зададе break, освен групата команди, отговаряща на намерения case-случай, ще се изпълнят и всички останали групи команди** след него. Изводът е, че трябва да се внимава каква точно логика трябва да се реализира и да не се забравя за съществуването на break.

Скрито от потребителите (програмистите), сравненията на case-стойностите от числов и изброим тип се извършват с оператора ==, а низовете се сравняват с метода equals на класа String,

От синтаксиса на конструкцията се вижда, че след „case стойност :" не е задължително да се задават команди. По този начин може да се опише една и съща група от команди да се отнася за няколко case-стойности.

Конструкцията switch-case може да се реализира и чрез множество вложени if-else конструкции. Когато броят на случаите е много голям, if-else конструкциите може да станат трудни както за описване, така и за четене.

Тривиален пример за представяне на конструкцията switch-case е работа с месеци. В следващия клас създаваме метод, който получава като параметър число от 1 до 12 за месеците и извежда името на месеца и тримесечието.

```
public class SwitchCaseStatement {
      public static void monthInfo(int monthNum){
            String monthName; // име на месец
            String quarter;   // тримесечие

            switch(monthNum){
                  case 1: monthName="януари"; quarter="първо"; break;
                  case 2: monthName="февруари"; quarter="първо"; break;
                  case 3: monthName="март"; quarter="първо"; break;
                  case 4: monthName="април"; quarter="второ"; break;
                  case 5: monthName="май"; quarter="второ"; break;
                  case 6: monthName="юни"; quarter="второ"; break;
                  case 7: monthName="юли"; quarter="трето"; break;
                  case 8: monthName="август"; quarter="трето"; break;
                  case 9: monthName="септември"; quarter="трето"; break;
                  case 10: monthName="октомври"; quarter="четвърто"; break;
                  case 11: monthName="ноември"; quarter="четвърто"; break;
                  case 12: monthName="декември"; quarter="четвърто"; break;
                  default: monthName="несъществуващ"; quarter="неизвестно";
            }

            System.out.println("Месец с номер " + monthNum
                        + " е " + monthName + " и е в "
                        + quarter +" тримесечие.");
      }

      public static void main(String[] args) {
            monthInfo(1);
            monthInfo(7);
            monthInfo(13);
      }
}

Резултат:
Месец с номер 1 е януари и е в първо тримесечие.
Месец с номер 7 е юли и е в трето тримесечие.
Месец с номер 13 е несъществуващ и е в неизвестно тримесечие.
```

Как да се справим с повторенията в кода

Това че променливата quarter за тримесечията на няколко места получава една и съща стойност не е добра реализация, защото при някое от изписванията може да се сгреши буква и тогава, тримесечията няма да са четири, а повече. Това може да доведе до допълнителни проблеми, ако в приложението се обработват тримесечията.

Едно от най-важните неформални правила в програмирането е „да не се дублира код”. Дублирането на код може да създаде множество проблеми, напр. при открита грешка на някое място или необходимост от промяна, е възможно да се пропусне някой фрагмент от дублирания код. Има различни механизми за избягване на повторения и те са свързани с познаването на синтаксиса и смисъла на всички езикови конструкции. Най-общо премахването на повторенията става чрез създаване на обобщение (генерализация), което може да има различни форми: константа, цикъл, метод, наследяване, абстракция и др.

В примера, който разглеждаме, за избягване на повторенията, вместо литерали за месеците и тримесечията може да се ползват предварително създадени константи. Друг подход, който използва switch-case конструкцията, е задаването на тримесечията да става едновременно за няколко case-стойности (в отделна конструкция):

```
switch(monthNum){
      case 1: case 2: case 3: quarter="първо"; break;
      case 4: case 5: case 6: quarter="второ"; break;
      case 7: case 8: case 9: quarter="трето"; break;
      case 10: case 11: case 12: quarter="четвърто"; break;
      default: quarter="неизвестно";
}
```

Цикъл

Цикълът (loop) **предоставя възможност за многократно изпълнение на** част от **кода.** Кодът, който се изпълнява многократно, се нарича **тяло на цикъла.**

Едно изпълнение на тялото на цикъл се нарича итерация.

Изпълнението на цикъла зависи от някакво **логическо условие**. Логическото условие е израз, в който обикновено има променлива, чиято стойност се променя по време на итерациите. **Параметър, от който зависи логическото условие, се нарича управляващ параметър.** Може да има няколко управляващи параметри. За да не стане цикълът безкраен, трябва условията да се задават правилно и внимателно да се управлява промяната на параметъра/параметрите. В някои случаи нарочно се прави безкраен цикъл и се използват операторите break и return за прекъсването му.

Преди да се изпълни една итерация, се изчислява стойността на логическия израз. Ако той има стойност true, тялото се изпълнява. Когато логическото условие получи стойност false, цикълът престава да се изпълнява.

Примери за дейности, при които може да се ползват цикли са:

- обхождане на списък (масив), изискващ еднотипна обработка на всеки отделен елемент – хора, фирми, числови (статистически) данни и много др.;
- обработка на елементите на низ, символ по символ;
- обхождане на елементите на матрица (таблица) – чрез два цикъла –външният цикъл обхожда таблицата по редове, а вътрешния – по стълбове;
- и др.

Циклите са два основни вида: с пред-условие и със след-условие (пост-условие). При циклите с пред-условие, условието се проверява преди изпълнението на итерацията, а при циклите с пост-условие, първо се изпълнява итерацията и след това се проверява условието (дали цикълът да приключи). Използват се конструкциите for loop, while loop, do-while loop за различните видове цикли.

Итератор е трети вид цикъл, при който няма явно зададени условия. Описва се с конструкцията for-each. Чрез цикъл-итератор може да обхождат всички елементи на списък. Реализира се чрез вътрешен (скрит) указател към следващ елемент. В началото указателят реферира първия елемент. При всяка итерация се работи с елемента, сочен от

указателя, а след изпълнението ѝ, указателят автоматично се пренасочва към следващия елемент. Цикълът приключва, когато указателят спре да сочи следващ елемент. Този вариант на цикъла е разгледан в частта Цикъл for(-each) за итериране върху елементите на масив на глава 8.

Цикъл с пред-условие while

Синтаксисът на цикъла while е:

```
while(<условие>){
   [<тяло>]
}
```

Може да се чете по следния начин:

```
Докато <условието> е истина, изпълнявай <тялото>!
```

Управляващият параметър се декларира преди тялото на цикъла, а стойността му се променя в тялото на цикъла.

В следващия пример се отпечатват числата от 1 до 5. Създадена е променлива i, която последователно приема стойностите [1, 2, 3, 4, 5]. При всяка итерация се отпечатва текущата стойност на i, след което се увеличава с единица. В условието за край като управляващ параметър се използва същата променлива i. Не е задължително управляващите променливи да са част от работната логика, изпълнявана в тялото на цикъла. В примера, употребата на управляващата променлива като работна (променлива) е случайна възможност, която използваме.

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

        // желаем да отпечатаме числата от 1 до 5
        int i=1; // на управляваща променлива i задаваме стойност 1

        while(i<=5){                // докато i е по-малко или равно на 5
            System.out.println(i); // отпечатваме i
            i++;                   // и увеличаваме i с единица
        }
    }
}
```

В резултат от изпълнението се извеждат числата от едно до пет на отделни редове. Може да се промени отпечатването им, като се изведат на един ред, разделени със запетаи. За целта след всички числа, без последното, с помощта напр. на условния оператор, се отпечатват запетайки:

```
System.out.print(i + (i<5?", ":""));

Резултатът в този случай е:
1, 2, 3, 4, 5
```

При цикълът while е възможно да не се изпълни нито една итерация, ако условието още в началото има стойност false. Например, ако началната стойност на i не се задава като литерал, а се получава или изчислява по друг начин, тя може да има стойност 10 например.

Цикъл със след-условие do-while

Синтаксисът на цикъла do-while е:

```
do{
   [<тяло>]
}while(<условие>);
```

Може да се чете по следния начин:

```
Изпълнявай <тялото>, докато <условието> е истина!
```

Символът ; (точката и запетая) за край на конструкцията е задължителен. Той не е задължителен за разгледания в предната част оператор за цикъл while.

Както и при цикъла while, управляващият параметър се декларира преди тялото на цикъла, а стойността му се променя в тялото (на цикъла).

Реализация на задачата за извеждане на числата от едно до пет с do-while изглежда по следния начин:

```
int i=1;
do{            // изпълнявай тялото...
      System.out.print(i + (i<5?", ":""));
      i++;
}while(i<=5); // ...докато i е по-малко или равно на 5
```

Разликата между цикъла със след-условие do-while и цикъла с пред-условие while е, че при do-while със сигурност ще се изпълни поне една итерация. Нека например i има начална стойност 10. Тялото ще се изпълни и ще се отпечата 10, след което ще се извърши за първи път проверка на условието, което ще има стойност false.

Цикъл с пред-условие for

Синтаксисът на цикъла for, в който с УП е означено „управляващи променливи” е:

```
for([<инициализация на УП>];[<условие>];[<промяна на УП>]){
   [<тяло>]
}
```

Логиката при цикъла for е същата като при while. Разликата е, че при for има заглавна (декларативна) част, в която има точно определени места за описание на основните дейности, свързани с управлението на цикъла:

- **Инициализация на управляващи променливи** – задават се начални стойности на управляващите променливи. Тази част се изпълнява еднократно в началото на изпълнението на конструкцията for.
- **Условие за изпълнение на тялото на цикъла –** Изчислява се преди всяко изпълнение на тялото на цикъла. Ако условието е истина, се изпълнява тялото на цикъла, в противен случай програмата приключва с изпълнението на цикъла и продължава със следващата команда.
- **Промяна на управляващи променливи** – в тази част се променя стойността на управляващите променливи. Изпълнява се след всяко изпълнение на тялото на цикъла.

Трите части, разделени помежду си с ; (точка и запетая) не са задължителни. Ако те не се опишат, цикълът става безкраен. В този случай може да се организира логиката за приключването му по друг начин – с командите break или return.

Чрез конструкцията за цикъл for се разделя логиката по управление на цикъла от работната логика. Управлението е изнесено в декларативната част, а работната логика е в тялото на цикъла.

Това, че местата за основните дейности в конструкцията for са точно определени, ни предпазва от случайни грешки. В показаните примери с while и do-while се променя стойността на управляващата променлива едва след като е изпълнена работната логиката – отпечатването ѝ, каквато е и стандартната логика при for. Случайно или нарочно би могло да се извърши промяна на стойността ѝ на друго – неправилно място.

Реализацията с for на вече разглежданата задача за извеждане на числата от едно до пет, е следната:

```
for(int i = 1; i <= 5; i++){
      System.out.print(i + (i < 5 ? ", " : ""));
}
```

В този пример, тъй като променливата i е декларирана в блока на цикъла for, тя ще е видима само в този блок. Ако е необходимо да е достъпна и след изпълнението на цикъла, трябва да се декларира над него.

В частта за инициализиране може да се задават стойности на няколко променливи, като отделните присвоявания са разделени със запетаи. Аналогично, в частта за промяна може да се променят стойностите на променливите. Това е показано в следващия примерен код:

```
for(int i = 1, j=20; i<j; i+=2, j-=2){
      System.out.print((j-i) + (i<j-4?", ":""));
}

Резултат:
19, 15, 11, 7, 3
```

Тук, в частта за инициализация, за променливата i е зададена стойност 1, а за j – 20. След всяка итерация първата стойност се увеличава с две, а втората се намалява с две.

Пример – извеждане на цели числа в указан интервал

В следващия пример се извеждат целите числата в интервала [n, m], където n и m са параметри на метод. Създадени и използвани са два метода – за показване в прав и обратен ред.

```
public class NumbersNtoM {
      // Показва числата от n до m (n<m) в прав ред
      public static void rightNumbers(int n, int m){
            for(int i=n; i<=m; i++){
                  System.out.print(i + (i < m ? ", " : "\n"));
            }
      }

      // Показва числата от n до m (n<m) в прав ред
      public static void reverseNumbers(int n, int m){
            for(int i=m; i>=n; i--){
                  System.out.print(i + (i > n ? ", " : "\n"));
            }
```

```
        }

        public static void main(String[] args) {
                rightNumbers(4, 12);
                reverseNumbers(5, 9);
        }
}

Резултат:
4, 5, 6, 7, 8, 9, 10, 11, 12
9, 8, 7, 6, 5
```

Особеност на методите за показване на числата е, че първият параметър трябва да е по-малък от втория. За да не се затрудняват потребителите на метода, по-добре е да им се позволи да задават произволни числа. За целта, преди циклите, ако е необходимо може да се извърши размяна на стойностите на променливите:

```
        if(n>m){                // ако n>m - трябва да ги разменяме
                int temp = n;   // записваме n във временна променлива temp,
                n = m;          // Тогава на n задаваме m,
                m = temp;       // а на m - n чрез temp
        }
```

Сума на числа в указан интервал

За намиране на сумата на целите числа в интервала [n, m] може да се използва цикъл (метода sum), в който се променя управляващата променлива i от n до m със стъпка 1. В началото сумата е 0, а при всяка итерация към нея се добавя стойността на i.

```
public class SumNumbers {
  // Сума на целите числата от n до m (n<m)
  public static void sum(int n, int m){
    int sum = 0;                // променлива за сума
    for(int i=n; i<=m; i++){    // променяме i от n до m със стъпка 1
       sum += i;                // при всяка итерация добавяме i към сумата
    }
    System.out.println("Сумата на числата от " + n + " до " + m + " е " + sum);
  }

  // Сума на елементите на аритметична прогресия
  public static void sumArithmeticalProgression(int n, int m){
    int sum = (m+n)*(m-n+1)/2; // (a0+an)*броя на елементите/2;

    System.out.println("Сумата на числата от " + n + " до " + m + " е " + sum);
  }

  public static void main(String[] args) {
    sum(1, 5);
    sumArithmeticalProgression(1, 5);
    sum(5, 8);
    sumArithmeticalProgression(5, 8);
  }
}

Резултат:
Сумата на числата от 1 до 5 е 15
Сумата на числата от 1 до 5 е 15
Сумата на числата от 5 до 8 е 26
Сумата на числата от 5 до 8 е 26
```

Както вече споменахме, „по-лесно” е да се намери сумата (метод sumArithmeticalProgression) като се отчете, че числата образуват аритметична прогресия с първи член n, последен – m, и брой на елементите (m-n+1). „По-лесно”, всъщност означава че алгоритъма за сума на аритметична прогресия е с константна сложност O(1), а сложността на цикъла е линейна: O(m, n) = (m-n+1)*3 (три са повтарящите се основни действия – сравнение, увеличаване с 1 и сумиране). Ако интервала, за който искаме да извършим сумиране е например [1, 100000], при изпълнение на цикъла ще се реализират 100000 итерации, което ще е доста по-бавно от намирането на произведение на две суми и делението им.

Друга разлика в двата алгоритъма е, че при големи числа, втория – sumArithmeticalProgression по-бързо ще предизвиква препълване на типа int, който се използва за сума. Докато при цикъла само се добавя текущата стойност на i, във втория случай първо се изпълнява произведение на две големи числа и едва след това те се делят – (m+n)*(m-n+1)/2. За да се направи вторият алгоритъм по-добър при работа с големи числа, може да се направим проверка кой от двата множителя (m+n) или (m-n+1) е четно число (единия със сигурност е четно число) и делението на 2 да се извърши с него и след това да се изчисли произведението с другия. Разбира се, и в двете решения може да се промени типа за сумата на long или BigInteger.

Произведение на цели числа в указан интервал

Произведението на последователни цели числа в указан интервал се реализира подобно на събирането – чрез цикъл. За него няма формула за бързо пресмятане.

Тук особеност е, че променливата за произведение първоначално се инициализира със стойност едно. При всяка итерация тя се умножава по текущата стойност на управляващата променлива.

В следващия пример е показано намирането на произведение на четните числа в интервала [n, m].

```
public class ProductNumbers {
  // Произведение на четните числата от n до m (n<m)
  public static void productEven(int n, int m){
    long product = 1;
    for(int i=n; i<=m; i++){  // променяме i от n до m със стъпка 1
      if(i%2==0){             // ако i се дели на 2...
        product *= i;         // го умножаваме с текущото произведение
      }
    }
    System.out.println("Произведението на четните числа от "
                       + n + " до " + m + " е " + product);
  }
  public static void main(String[] args) {
    productEven(1, 5);
    productEven(4, 9);
  }
}

Резултат:
Произведението на четните числа от 1 до 5 е 8
Произведението на четните числа от 4 до 9 е 192
```

В тялото на цикъла се обхождат всички числа от n до m. За всяко число се проверява дали е четно, и само ако е четно се умножава с намереното до текущата итерация произведение.

Може да се реализира и друга логика: стъпката на цикъла да е през две четни числа и въобще да няма проверка за четност в тялото му. За да стане това, достатъчно е първото число, с което се инициализира управляващата променлива, да е четно. В този случай сложността на алгоритъма ще си остане линейна, но ще се изпълнят двойно по-малко дейности. Новата реализация е показана в следващия код:

```
      public static void productEvenQuick(int n, int m){
            long product = 1;
            if(n%2!=0){  // ако n е нечетно...
                  n++;   // ...става четно
            }
            for(int i=n; i<=m; i+=2){ // стъпка 2
                  product *= i;
            }
            System.out.println("Произведението на четните числа от "
                                    + n + " до " + m + " е " + product);
      }
```

Вложени цикли

В тялото на един цикъл може да се описват различни други конструкции – if, switch, изрази, цикли и др.

Вложени цикли има, когато в тялото на един външен цикъл се изпълнява друг, вътрешен цикъл. При всяка една итерация на външния цикъл, вътрешният изпълнява всичките си итерации. Например, ако външният цикъл има 10 итерации, а вътрешният 5, то тялото на вътрешния ще се изпълни общо 50 (=10*5) пъти.

Чрез вложени цикли може да се реализират различни алгоритми за сортиране, да се обхождат матрици и др.

Следващият пример представя таблицата за умножение на числата от 1 до n. Управляващата променлива на външния цикъл се променя от 1 до n. За всяка от стойностите ѝ по време на итерация, управляващата променлива на вътрешния цикъл също се променя от 1 до n.

```
public class NestedLoop {
  public static void multiplicationTable(int n){
    System.out.println("Таблица за умножение на числата от 1 до n - с повторения");
    for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++){
        System.out.println(i + " * " + j + " = " + i*j);
      }
    }
  }

  public static void main(String[] args) {
    multiplicationTable(3);
  }
}

Резултат:
Таблица за умножение на числата от 1 до n - с повторения
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

Както се вижда в резултата, някои от произведенията се повтарят, но с разменени места на множителите. Повторенията може да се премахнат. Повтарят се произведенията 2*1, 3*1, 3*2. Тази закономерност може да се обобщи : това са случаи, при които стойностите на j са по-малки от i. Следователно, решението е във вътрешния цикъл, j да се променя от i (а не от 1) до n.

```
public static void multiplicationTableNoRepeats(int n){
  System.out.println("Таблица за умножение на числата от 1 до n - без повторения");
  for(int i=1; i<=n; i++){
       for(int j=i; j<=n; j++){
             System.out.println(i + " * " + j + " = " + i*j);
       }
  }
}

Ако метода се стартира за n=3 резултата е:
Таблица за умножение на числата от 1 до n - без повторения
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 2 = 4
2 * 3 = 6
3 * 3 = 9
```

Пример за създаване на програма с конзолно меню

Често използван подход при конзолните приложения е управлението на програма от меню. Менюто описва възможните действия и предлага код (един или повече символа), чрез въвеждането на който може да се стартира всяко действие. В следващия пример е показано избор на няколко продукта от магазин.

```
import java.util.Scanner;

public class ShopConsoleMenu {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    String choice;  // избрана опция от менюто
    double price=0; // цена на продуктите

    // в цикъл многократно показваме менюто
    do{
       System.out.print("Изберете (1-Добавяне на продукт; 2-Плащане; 3-Изход): ");
       choice = scanner.next();  // прочита се въведения избор
       switch(choice){
         case "1": price = addToCart(scanner); break; // извикване на метод
         case "2": System.out.println("Цената е " + price + " лв."); break;
         case "3": System.out.println("Довиждане!"); break;
         default: System.out.println("Въвели сте грешен избор!");
       }
    }while(!choice.equals("3"));
    // цикълът се изпълнява докато от конзолата не се въведе 3

    scanner.close();
  }

  // Метод за добавяне в кошницата на продукти
  // Получава Scanner - за да не се създава многократно
  // Връща цената на добавените продукти
  public static double addToCart(Scanner scanner){
    String choice;
    double price=0;

    // При избор на продукт увеличаваме дължимата сума
    do{
```

```
        System.out.print("1-Хляб(1 лв.); 2-Вода(0.80 лв.); 3-За сега спирам...: ");
        choice = scanner.next();
        switch(choice){
          case "1": price+=1; break;
          case "2": price+=0.8; break;
        }
      }while(!choice.equals("3"));

      return price;
    }
}
```

Примерен резултат:

```
Изберете (1-Добавяне на продукт; 2-Плащане; 3-Изход): 1
1-Хляб(1 лв.); 2-Вода(0.80 лв.); 3-За сега спирам...: 1
1-Хляб(1 лв.); 2-Вода(0.80 лв.); 3-За сега спирам...: 2
1-Хляб(1 лв.); 2-Вода(0.80 лв.); 3-За сега спирам...: 3
Изберете (1-Добавяне на продукт; 2-Плащане; 3-Изход): 2
Цената е 1.8 лв.
Изберете (1-Добавяне на продукт; 2-Плащане; 3-Изход): 3
Довиждане!
```

Следват кратки разяснения за реализацията. Менюто се изпълнява поне веднъж, в цикъл (do-while). В него:

- Извеждат се възможните опции „добавяне на продукт – плащане – изход";
- Въведеният избор се прочита със Scanner;
- В зависимост от избора, в switch се изпълнява нужното действие. При едно от действията (добавяне на продукт), се извиква друг метод с под-меню.

Примерът описва по формален и опростен начин възможно обслужване на купувач в магазин. Недостатък е, че продуктите в кошницата не се записват в програмата и съответно не могат да се извадят от нея. Това може да се организира с помощта на класове за описание на продуктите и структури за съхранението им в списъци.

Команди за прекъсване break, continue и return

Командите break, continue и return прекъсват изпълнението на основната работна логика:

- **continue** – **прекъсва текущата итерация** на цикъл. Ако в тялото на цикъла има описани команди, след мястото на извикването на continue, те не се изпълняват. Изпълнението на цикъла продължава със следващата итерация.
- **break** – използва се за прекъсване на **цикъл и switch** конструкции. След извикването му, програмата продължава да се изпълнява със следващата след конструкцията команда.
- **return** – прекъсва изпълнението на **метод**, но **в частност и на цикъл**. След изпълнението му, програмата продължава да се изпълнява от мястото на извикване на метода. Ако метода връща стойност, след return трябва да има израз или стойност от съответния тип.

Обикновено прекъсване е необходимо при възникване на някакво специално условие, проверявано в if конструкция или ако трябва да се опростят няколко сложни if-else конструкции.

Действието на командите за прекъсване са показани с демонстрационни примери. Някои от тях могат да бъдат реализирани и по други, по-подходящи начини.

Нека разгледаме задачата за отпечатване на броя на целите числа от даден интервал [n, m], които не се делят на друго число divisor. В цикъл (метод countNumbers) се проверява дали всяко число от интервала се дели на divisor. Ако се дели, не се прави нищо в текущата итерация, в противен случай стойността на някакъв брояч се увеличава

с 1. Вместо проверката да се реализира в if-else конструкция, при което тялото на if е празно, може да се използва continue по показания в следващия пример начин.

```
public class ContinueStatement {
	// Връща брой на числата от n до m, които не се делят на divisor
	public static int countNumbers(int n, int m, int divisor){
		int count = 0;			// в началото броят е нула

		for(int i=n; i<=m; i++){
			if(i%divisor==0){ // Ако i се дели на divisor...
				continue;	// ... приключваме текущата итерация
			}				// ... иначе - без да записваме else...

			count++;			// ... увеличаваме текущия брой с 1
		}

		return count;			// ... връщаме броя
	}

	// Извежда информация за брой на числата от n до m,
	// които не се делят на divisor - извиква countNumbers
	public static void countNumbersInfo(int n, int m, int divisor){
		int count = countNumbers(n, m, divisor);

		System.out.println("Броят на числата в интервал ["
				+ n + ", " + m + "], които не се делят на "
				+ divisor + " е " + count);
	}

	public static void main(String[] args) {
		countNumbersInfo(3, 30, 11);
		countNumbersInfo(3, 30, 5);
	}
}
```

Резултат:

```
Броят на числата в интервал [3, 30], които не се делят на 11 е 26
Броят на числата в интервал [3, 30], които не се делят на 5 е 22
```

Основната работна логиката в тялото на цикъла, (в случая тя е само да се увеличи брояча с единица), се изпълнява извън тялото на условния оператор.

В следващия пример, търсим най-малкото естествено число от интервала [n, m], което се дели на divisor. Зададено е изкуствено изискването числата да са естествени заради създадения метод, който желаем да връща намереното число или нула, ако не е намерено такова. При обхождането на числата (метод searchNumber), ако текущото число се дели на divisor, търсенето приключва. За търсено число се приема текущото, цикълът приключва с break и методът връща намереното число.

```
public class BreakReturnStatement {
	// Връща най-малкото число в интервала [n, m] (n>0 и m>0),
	// което се дели на divisor или 0 - ако не е намерено
	public static int searchNumber(int n, int m, int divisor){
		int search = 0;			// в началото търсеният е нула

		for(int i=n; i<=m; i++){	// обхождаме всички
			if(i%divisor==0){	// Ако i-ия се дели на divisor...
				search = i;	// няма смисъл да търсим повече и...
				break;		// ... излизаме от цикъла
			}
```

```
            }

            return search;            // връщаме намереното число
        }

        // Извежда инф. за най-малкото число в интервала [n, m] (n>0 и m>0),
        // което се дели на divisor - използва searchNumber
        public static void searchNumberInfo(int n, int m, int divisor){
            int search = searchNumber(n, m, divisor);

            if(search>0){
                System.out.println("Най-малкото число в интервал ["
                        + n + ", " + m + "], което се дели на "
                        + divisor + " е " + search);
            }else{
                System.out.println("Числото " + divisor
                        + " не е делител в интервал [" + n + ", " + m + "]");

            }
        }

        public static void main(String[] args) {
            searchNumberInfo(20, 50, 11);
            searchNumberInfo(41, 50, 5);
            searchNumberInfo(1, 50, 51);
        }
}

Резултат:
Най-малкото число в интервал [20, 50], което се дели на 11 е 22
Най-малкото число в интервал [41, 50], което се дели на 5 е 45
Числото 51 не е делител в интервал [1, 50]
```

Не е необходимо да се обхождат числата, ако divisor е по-голямо от m. Може самостоятелно да реализирате такава проверка.

Вместо break може да се използва return, както е показано в следващия примерен метод:

```
public static int searchNumberReturn(int n, int m, int divisor){
        for(int i=n; i<=m; i++){
            if(i%divisor==0){     // Ако i-ия се дели на divisor...
                return i;      // връща намереното число и излиза от метода
            }
        }

        return 0;    // Ако цикъла е приключил, значи не е намерено число
}
```

Въпроси и задачи за упражнения

1. Създайте и използвайте метод, който намира и връща минималното от две цели числа от тип int, зададени като параметри.
2. Създайте програма за намиране на минималното от няколко – от 3 до 6 числа, въвеждани по време на изпълнение на програмата.
3. Сравнете резултатите на методите sum и sumArithmeticalProgression в частта за Сума на числа в указан интервал. При какви стойности и защо престават да изчисляват коректно? Променете метода sumArithmeticalProgression така, че резултатите да са идентични с резултатите на sum при по-големи числа.
4. Създайте програма за намиране на сумата (като int) на нечетните цели числа в интервал [n, m], като реализирате два алгоритъма: с цикъл и с формулата за сума на аритметична прогресия. Използвайте BigInteger за типа на параметрите.

5. Изведете ‘пирамида’ с n реда от символи, увеличаващи се с един на всеки следващ ред:
   A
   AA
   AAA
6. Изведете кодовете и символите на българските букви в кодова таблица UNICODE.
7. Изведете ‘пирамида’ от символи с n реда и начален символ symbol. За всеки символ съседите му отдясно и отдолу са с код по-голям с единица. Напр.
   A
   Б В
   В Г Д
   Г Д Е Ж
8. Създайте и използвайте метод за намиране на n факториел.
9. Поправете логиката в примера за управление на магазин с конзолно меню:
   а. Ако цената на продуктите е по-голяма от 0, не може да се напусне магазина;
   б. Ако цената е 0 лв., при избор на плащане се извежда съобщение “Кошницата Ви е празна!”.
10. Създайте програма, обслужваща банкомат, която поддържа текущата налична сума и PIN кода на 3 електронни дебитни карти. При стартиране на програмата се визуализира меню със следните възможности: 1. Теглене на сума; 2. Проверка на наличност; 3. Промяна на ПИН код; 4. Добавяне на сума; 5. Изход.
11. Създайте програма, в която се въвеждат параметри a и b за формулата за права в равнината y=a*x+b. При въвеждане на интервал [x1, x2], отпечатайте стойностите на y за x, изменящо се в границите на въведения интервал със стъпка 0.1.
12. Създайте програма за отпечатване стойностите на парабола $y=a*x^2+b*x+c$ при зададени a, b, c и интервал [x1, x2] за промяна на x със стъпка 0.1.
13. За дадено число намерете броя на цифрите му.
14. За дадено естествено число n намерете обърнатото число. Например, обърнатото на 1234 е 4321.
15. Намерете дали дадено естествено число е палиндром, т.е. дали то е равно на обърнатото му число. Напр. 123321 и 123321 са палиндроми.
16. Проверете дали зададена високосна година year е високосна.
17. За две цели положителни числа a и b намерете най-големия общ делител.
18. За две цели положителни числа a и b намерете най-малкото общо кратно.

Упътвания към задачите

1. Използвайте примера за намиране на максимално число от част Условна конструкция if-else.
2. Използвайте вече създадения в задача 1 метод.
3. Променете формулата за намиране на сума в sumArithmeticalProgression, така че първо да се извършва делението.
4. Използвайте задача 3.
5. Използвайте два вложени цикъла: външен за редове и вътрешен за брой букви.
6. Българските букви в UNICODE са с номера от 1040 до 1103. Използвайте цикъл и преобразуване по тип за показването им.
7. Вижте 5-та и 6-та задачи.
8. Вижте примера в частта Произведение на числа в указан интервал и дефиницията на факториел.
9. В съответните случаи („Изход” и „Плащане”) използвайте оператор if-else.
10. Използвайте логиката на задача 9.

11. Създайте метод void y(double a, double b, double x), който изчислява и отпечатва стойността на израза a*x+b. В основния main метод задайте a, b, x1, x2, а след това в цикъл за x, променящо се от x1 до x2 със стъпка 0.1, изпълнявайте многократно метода y.
12. Вижте предходната задача.
13. В цикъл използвайте брояч за броя пъти, което едно число n=n/10 може да се раздели на 10 преди да стане равно на 0.
14. Използвайте операторите / и % в цикъл. С n%10 намираме последната цифра, а с n=n/10 получаваме число, което е с един разряд по-малко и при което липсва последната цифра. Във всяка итерация новообразуващото се число, което в началото има стойност нула, умножавате по 10 и към него прибавяте остатъка от целочисленото деление на 10.
15. Вижте предходната задача.
16. Създайте метод, получаващ като параметър година, който връща boolean стойност. Една година е високосна, ако се дели на 4 без остатък. Има изключение – годините, кратни на 100 не са високосни, с изключение на годините кратни на 400, които са високосни.
17. Потърсете в Интернет информация за алгоритъма на Евклид за намиране на най-голям общ делител.
18. Потърсете в Интернет информация за алгоритъма.

Глава 8. Едномерни масиви

Масивите са сложни обекти, съдържащи множество от елементи. Казва се, че те описват **колекция от елементи.** Всеки елемент се представя като двойка величини „ключ-стойност" (key-value). **Ключовете** в един масив **са уникални**, а стойностите може да се повтарят. Чрез името на променливата и стойността на ключа може да се достъпи съответната на ключа стойност.

В масиви може да се описва информация за хора, фирми, населени места, измервания и всякакви други обекти от реалния свят.

Според типа на ключовете, масивите могат да бъдат:

- **Обикновени** – определят се като **последователност от еднотипни елементи.** Ключовете са последователни цели числа, а стойностите на масива са от един и същи тип. **Ключовете** в този случай **се наричат индекси,** като първият индекс има стойност 0, следващият – 1 и т.н.
- **Асоциативни** – ключовете и стойностите може да са от произволен тип.

В различни езици за програмиране са реализирани различни видове масиви. **В Java има само обикновени масиви**, но има и стандартни класове за други видове типове за описание на колекции – списъци, множества, хеш-таблици и др.

Според броя на размерностите, масивите са:

- **едномерни** – за достъп до една стойност се използва един ключ;
- **двумерни** – за достъп до една стойност се използва комбинация от два ключа. Използват се за представяне на матрици;
- **многомерни** – може да се използват за представяне на йерархични структури.

За обхождането и обработката на масиви се използват цикли.

В настоящата глава са разгледани начините за създаване и стандартни методи за обработка на масиви.

Масиви в Java

Масивите в Java са обекти от невидими за потребителя класове. С помощта на езикови конструкции класовете се скриват, а с масивите се работи по относително стандартен за всички езици за програмиране начин.

В Java се използват обикновени масиви. Те са последователност от еднотипни елементи. Индексите са последователни цели числа, започващи от 0, а стойностите са от един и същи тип.

Масивите се декларират по начин, подобен на другите променливи:

```
<тип> [] <име на променлива за масива>;
```

Използването на правоъгълни скоби ([]) ги отличава от другите променливи, представящи единични стойности. Правоъгълните скоби може да се записват пред и след името на масива. Важно е да се отбележи, че при едновременна декларация на няколко променливи, когато скобите се указват след типа, всички следващи променливи се създават като масиви, а когато скобите са след името на променлива, те се отнасят само за нея. Напр. по следния начин може да се декларират няколко едномерни масива:

```
int [] intArray;         // масив от цели числа
double [] doubleArray;   // масив от реални числа
String [] names;         // масив с имена - низове
Person [] persons;       // масив с обекти от клас Person
```

Размерностите на масивите се определят от използваните двойки правоъгълни скоби: двумерен масив се записва с две двойки, тримерен – с три и т.н. Например:

```
int [][] matrix;        // двумерен масив - матрица
double [][][] array3;   // тримерен масив
```

Променливите за масиви са референции, подобно на обектите. Затова при декларация те имат стойност null.

За **създаването на масив** се използва операторът new, при което се определя и броят на елементите на масива:

```
new <тип> [<брой на елементите на масива>];
```

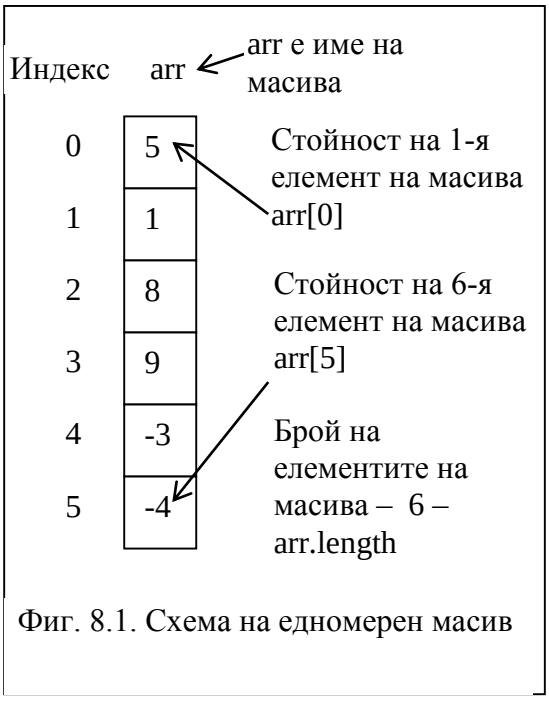
Например,

```
intArray = new int[5];       // масив с 5 елемента от тип int
doubleArray = new double[4];
names = new String[6];

int numPersons = 3;
Person [] persons = new Person[numPerson];
matrix = new int[4][3]; // матрица с 4 реда и 3 колони – 12 елемента
```

Масивите се създават по време на изпълнение на програмата в динамичната памет. Броят на елементите на масив не може да се променя динамично, но една променлива за масив може да реферира различни масиви по време на работа на програмата.

При създаването на масив, за елементите на масива се задават **стойности по подразбиране** от съответния тип. **За примитивните типове – целочислени, реални и булев се задават съответно 0, 0.0 и false, а за сложните типове – null.**

Едномерни масиви



Фиг. 8.1. Схема на едномерен масив

Всеки отделен елемент на масив се **достъпва с помощта на името на променливата на масива и в правоъгълни скоби се записва индекс – arr[0], arr[1].** При обръщение към несъществуващ индекс се получава изключение.

Броят на елементите на масива може да се вземе с помощта на **полето length**, достъпно чрез променливата за масив, напр., arr.length. Това улеснява обхождането в цикъл, при което управляващата променлива се променя от 0 до length-1 със стъпка 1.

С елементите на масива може да се работи по същия начин, както с обикновена променлива. Може да им се задава стойност чрез оператора за присвояване или да участват в изрази.

```
arr[0] = 5;         // задаване на стойност на първия елемент
arr[1] = arr[0]*6; // задаване на стойност на втория елемент
```

Удобен начин за инициализация на масив е чрез задаване на списък от стойности, разделени със запетая и оградени във фигурни скоби. Това може да се извършва само при декларация на променливата за масив.

```
int [] arr = {5, 1, 8, 9, -3, -4};
String [] names = {"Иван", "Мария", "Петър", "Петя"};
```

В примера се заделя памет за масив от 6 елемента от тип int и се инициализира със стойностите, описани във фигурните скоби. Аналогично се създава и масив от низове, съдържащ имена на хора. При това не е необходимо изрично да се използва операторът new и да се указва броят на елементите.

Въвеждане на масив от конзолата

За въвеждане на броя и стойностите на елементите на масив може да се използва класът java.util.Scanner. В следващия пример първо от конзолата се задава броят на елементите, а след това се създава масив от цели числа с указания брой елементи. В цикъл се дефинира управляваща променлива (i) с начална стойност 0, която се ползва за обхождане на елементите на масива. При всяка итерация стойността ѝ се увеличава с единица, а цикълът приключва, когато стойността ѝ стане по-голяма от броя на елементите-1, което е означено в кода по следния начин „i < a.length”.

```
import java.util.Scanner;

public class ConsoleArray {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Въведете брой на елементите на масива:");
    int num = scanner.nextInt();   // променлива за брой на елементите

    int a[] = new int[num];        // масив с num елемента
    System.out.println("Въведете елементите на масива:");
    for (int i = 0; i < a.length; i++) {
      System.out.print("a[" + i + "]=");
      a[i] = scanner.nextInt();   // въвеждане на i-тия елемент
    }

    scanner.close();
  }
}

Примерен вход-изход в конзолата:
Въведете брой на елементите на масива:3
Въведете елементите на масива:
a[0]=1
a[1]=2
a[2]=3
```

Генериране на масив от случайни числа

За да се тества как работи алгоритъм със случайни данни, понякога е необходимо да се генерират масиви със случайни числа, вместо да се въвеждат ръчно. Генерирането на

случайни числа се извършва с класа java.util.Random, както е показано в следващия пример:

```
import java.util.Random;

public class RandomArray {
  public static void main(String[] args) {
    Random rand = new Random();

    // брой на елементите - случайно число в инт. [5, 10]
    int num = rand.nextInt(6)+5;

    int a[] = new int[num];
    System.out.println("Задаване елементите на масива:");
    for (int i = 0; i < a.length; i++) {
      a[i] = rand.nextInt();   // генериране на случайна стойност...

      // ... и извеждане на информация за i-тия елемент
      System.out.println("a[" + i + "]=" + a[i]);
    }
  }
}

Примерен изход в конзолата:
a[0]=-1190354341
a[1]=766200137
a[2]=1104182079
a[3]=167567934
a[4]=-1768480890
```

Може да се генерира масив от цели числа, съдържащ стойности в указан интервал, с помощта на методите ints(), longs() и doubles(). Тези методи са въведени от версия 8 на Java. Примери за това са представени в глава 6. Тук по-подробно е представен един начин за създаване на случаен масив от цели числа:

```
import java.util.Random;
import java.util.Arrays;
import java.util.stream.IntStream;

public class IntStreamArray {
  public static void main(String[] args) {
    Random rand = new Random();

    // генериране на масив с 10 int числа в интервал от -10 до 20
    // методът ints() на Random връща обект-поток от тип IntStream
    IntStream is = rand.ints(10, -10, 21);

    // методът toArray() на IntStream връща масив
    int [] intArray = is.toArray();

    // методите toString(масив) на класа Arrays връщат масива като низ
    String arrayAsString = Arrays.toString(intArray);
    System.out.println(arrayAsString);
  }
}

Примерен резултат:
[14, -7, 15, 17, 12, -10, 2, 7, -3, -2]
```

Клас за работа с масиви – java.util.Arrays

Класът Arrays предоставя множество стандартни методи за работа с масиви като:

- запълване на масив с еднакви или случайно генерирани стойности – методи fill(), setAll(), parallelSetAll();
- копиране на масив или част от масив в нов масив – copyOf, copyOfRange;
- сортиране – sort(), parallelSort();
- двоично търсене – binarySearch();
- сравнение на два масива – equals();
- намиране на хеш код за масив, базиран на стойностите на елементите;
- преобразуване в низ – методи toString(<масив>) и др.

Някои от споменатите методи са реализирани във версия 8 на Java. Такива са методите за паралелно задаване на стойности, паралелно сортиране, работа с потоци и др.

В следващите примери са използвани някои от стандартните методи на класа Arrays, но основната цел е да покажем индивидуални реализации на логиката им.

Отпечатване на елементите на масив

В следващия пример, в клас IntArrayUtils е създаден метод, подобен на Arrays.toString(), както и метод за генериране на масив. В класа се работи само с масиви с елементи от тип int. Ако желаем да работим с други примитивни типове, трябва да се създаде друг подобен клас – като се копират методите и се смени типът int с желания друг тип.

```
import java.util.Random;
import java.util.stream.IntStream;

// Клас с помощни методи за работа с масиви с елементи от тип int
public class IntArrayUtils {
  // Метод, който генерира случаен масив с length елемента
  // от тип int в интервал [n, m]
  public static int[] genRandom(int length, int n, int m){
    Random rand = new Random();
    IntStream is = rand.ints(length, n, m+1);
    int [] intArray = is.toArray();
    return intArray;
  }

  // Метод, връщащ като низ масива, получен като параметър
  public static String toString(int [] array){
    // В stringArray ще добавяме елементите на низа,
    // оградени във фигурни скоби
    String stringArray = "{";

    // При всяка итерация добавяме i-тия елемент и запетая след него,
    // но само ако (i-тия елемент) не е последен
    for (int i = 0; i < array.length; i++) {
      stringArray += array[i] + (i < array.length-1?", ":"");
    }

    stringArray += "}";

    return stringArray;

  }
```

```
  public static void main(String[] args) {
    int [] arr = genRandom(5, 20, 30); // 5 случайни числа в инт. [20, 30]
    System.out.println(toString(arr)); // отпечатваме масива


    // Без да създаваме променлива за масив - генерираме и отпечатваме
    System.out.println(toString(genRandom(5, -5, 5)));
  }
}

Примерен резултат:
{26, 24, 20, 27, 24}
{2, -3, -5, 4, -4}
```

Може да се използват създадените методи в други примери:

- int[] genRandom(int length, int n, int m) – създава и връща масив с length на брой елемента от тип int. Стойностите им са в интервала [n, m].
- String toString(int [] array) – връща масива, подаден като параметър, във вид на низ.

Цикъл for(-each) за итериране върху елементите на масив

С модификация на цикъла for, по-известна в други езици за програмиране като for-each, може да се обхождат последователно всички елементи, без да е необходимо да се използват индексите на масива. По подобен начин се обхождат и други колекции от елементи в Java, като списъци, хеш таблици, множества и др.

Синтаксисът на цикъла for в този случай е:

```
   for (<променлива за елемент на масива>:<масив>){
      <тяло>
   }
```

В скобите на заглавната част на цикъла се задава променлива, в която при всяка итерация се записва следващата стойност на елемент на масива, който се указва след двете точки. В следващия пример се отпечатва масив с тази разновидност на цикъла for. Променливата value при всяка итерация приема следващата стойност на елемент от масива array.

```
public class ForEachExample {
      public static void main(String[] args) {
            int array[] = {1, 2, 3};

            for (int value:array) {
                  System.out.println(value);
            }
      }
}

Резултат:
1
2
3
```

Този вариант на цикъла не е подходящ за отпечатване на елементите, разделени със запетаи, защото не може да се направи проверка за последен елемент. Може да се добави променлива-брояч на елементите, но това би било разновидност на стандартния вариант на цикъла for – с управляваща променлива.

Предаване на масиви като параметри и променлив брой аргументи на метод

Параметрите на методите се описват така, както се описват декларации на самостоятелни променливи – обикновени или масиви. В определени случаи, при извикване на методи може да се очакват различен брой параметри. Може да се използват масиви или оператор „...” (три точки), указващ променлив брой параметри от даден тип. Ако методът притежава и други параметри, те трябва да са описани в началото на списъка с параметри. В следващия пример е деклариран метод varArgs(int ... ints), който получава променлив брой параметри от тип int в променливата ints. Променливата ints може да се обхожда като обикновен масив. Най-голямото предимство на променливия брой аргументи е възможността фактическите параметри да се задават като списък от различни променливи, или като масив, което е показано в main-метода в следващия пример.

```
public class VarArgsExample {
      // Метод който може да получава променлив брой параметри
      // от тип int в променлива ints
      public static void varArgs(int ... ints){
            // ints се обхожда като масив
            for (int i = 0; i < ints.length; i++) {
                  System.out.print(ints[i] + (i < ints.length-1?", ":""));
            }
            System.out.println();
      }
      public static void main(String[] args) {
            varArgs(1, 2);      // извикване с два параметъра
            varArgs(1, 2, 3);   // с три параметъра
            varArgs(new int[]{3, 4, 5}); // с масив от цели числа
      }
}

Резултат:
1, 2
1, 2, 3
3, 4, 5
```

Основни алгоритми за обработка на масиви

След като показахме как се създават едномерни масиви от числа и как могат да се отпечатват, ще разгледаме различни стандартни алгоритми за обработката им. Такива са:

- намиране на сума и средно-аритметично;
- намиране на минимален и максимален елемент;
- търсене на елемент по стойност;
- сортиране;
- двоично търсене (в сортиран масив) и др.

Повечето алгоритми се използват и при работа с не-числови масиви. Алгоритмите са описани в методи, получаващи масива като параметър. За коректната работа на методите трябва да се отчете фактът, че променливата за масив може да има стойност null и да се извърши подходяща проверка. Това не е направено, за да не се усложнява основната логиката.

Сума на елементите на масив

За да се намери сумата на елементите на масив, масивът може се обходи в цикъл и при всяка итерация текущия елемент да се добавя в променлива за сума. В примерната реализация е използван цикъл for-each, а за създаване и отпечатване на резултата - статичните методи, разработени в класа IntArrayUtils.

```
public class SumArray {
  // Метод за сума на елементите на масив
  public static int sum(int arr[]) {
    int sum = 0;      // в началото сумата е 0
    for (int i:arr){ // при всяка итерация
      sum += i;       // към сумата добавяме текущия ел.
    }
    return sum;       // връщаме сумата
  }

  public static void main(String[] args) {
    // генерираме масив с 5 елемента със стойности от 0 до 7
    int array[] = IntArrayUtils.genRandom(5, 0, 7);
    // Взимаме масива като низ
    String arrayToString = IntArrayUtils.toString(array);
    // изчисляваме сумата
    int sum = sum(array);
    // отпечатваме информация
    System.out.println("Сума на масива " + arrayToString + ": " + sum);
  }
}

Примерен резултат:
Сума на масива {7, 1, 6, 0, 6}: 20
```

Средно-аритметично на елементите на масив

Средно-аритметична стойност се получава, като се раздели сумата от елементите на масива на техния брой. Средно-аритметичната стойност може да не е цяло число. За да не се получи цяло число при делението на две цели (сума/брой), едното цяло число се преобразува до реално и чак след това се извършва делението.

```
public class AverageArray {
  // Метод за средно-аритметично на елементите на масив
  // Връща double - сумата/броя на елементите
  public static double average(int arr[]) {
    int sum = 0;      // в началото сумата е 0
    for (int i:arr){ // при всяка итерация
      sum += i;       // към сумата добавяме текущия ел.
    }
    return (double)sum/arr.length;     // ср.ар.
  }

  public static void main(String[] args) {
    int array[] = IntArrayUtils.genRandom(5, 0, 7);
    String arrayToString = IntArrayUtils.toString(array);
    double average = average(array);
    System.out.println("Ср. ар. на масива "+arrayToString+ ": " + average);
  }
}
Примерен резултат:
Ср. ар. на масива {2, 2, 1, 0, 2}: 1.4
```

В класа SumArray вече е създаден метод за сума. В случаи като този е препоръчително да се ползва готовият метод, вместо да се създава нов метод, включващ вече реализирана логика.

```
      public static double average(int arr[]) {
            return (double)SumArray.sum(arr)/arr.length;
      }
```

Тъй като методите за обработка на масиви са логически свързани, е добре да са организирани в общ клас-библиотека, което читателят може да направи самостоятелно.

Търсене на елемент по зададена стойност

Задачата при търсенето е да се провери дали дадена стойност се намира в масив. Методът може да връща булева стойност (true, false) или индекс на масива, в зависимост от това, как точно е формулирана задачата. Повече информация носи метод, връщащ индекса на намерения елемент. Ако не е намерен елемент, методът може да връща например числото -1, което не е валиден индекс.

Ако е необходимо методът да връща индекс, трябва да се отчете фактът, че в масива може да има няколко елемента с търсената стойност. Тогава методът може да връща индекса на първия намерен елемент, на последния или масив с всички намерени индекси.

Тук е реализирана задачата за извеждане на последния намерен индекс. Показано е обхождане на масив отзад напред. Ако масивът е неподреден, в процеса на търсене последователно се обхождат всички елементи (в случая отзад напред). Текущата стойност на масива се сравнява с търсената стойност и при съвпадение, методът приключва, като връща намерения индекс. Ако масивът е подреден, има по-бърз алгоритъм за двоично търсене, който е разгледан в следваща част.

```
public class SearchArray {
  // Метод за търсене на searchVal в масив arr
  public static int search(int arr[], int searchVal) {
    // Обхождаме масива, например, отзад напред
    for (int i = arr.length-1; i>=0; i--){
      if(arr[i]==searchVal){ // ако елемента е намерен
        return i;            // връщаме позицията му
      }
    }
    return -1;    // връщаме -1, ако не е намерен
  }

  public static void main(String[] args) {
    int array[] = IntArrayUtils.genRandom(5, 0, 7);
    String arrayToString = IntArrayUtils.toString(array);
    int searchVal = 3;                          // търсена стойност
    int searchPos = search(array, searchVal); // намерена позиция
    System.out.println("В масива " + arrayToString
      + " елемент " + searchVal
      + (searchPos==-1?" не е намерен":" е намерен на позиция " +
searchPos));
  }
}
```

Примерни резултати:

```
В масива {6, 4, 6, 4, 6} елемент 3 не е намерен
В масива {0, 3, 6, 5, 3} елемент 3 е намерен на позиция 4
```

Намиране на минимален и максимален елемент в масив

При търсене на минимален или максимален елемент трябва да се обходят всички елементи на масива.

Логиката на алгоритмите за намиране на минимален елемент е следната: в променлива за минимален елемент се записва стойността на първия елемент на масива. В цикъл се обхождат всички останали елементи. При всяка итерация, текущият елемент се сравнява с минималния. Ако текущият елемент е по-малък от минималния, то той става минимален, като се присвоява на променливата за минимален елемент. При приключване на цикъла, променливата за минимален елемент съдържа търсената стойност.

В следващия пример методът за намиране на минимален елемент е реализиран чрез цикъла for-each, а методът за максимален – чрез стандартния цикъл for.

```
public class MinMaxArray {
	// Метод, връщащ минималния ел. на масив arr
	public static int min(int arr[]) {
		int min = arr[0];	// първия ел. е минимален
		for (int val:arr){
			if(val<min){	// ако текущия е по-малък от минималния
				min=val; // той става минимален
			}
		}
		return min;		// връщаме min елемент
	}

	// Метод, връщащ максималния ел. на масив arr
	public static int max(int arr[]) {
		int max = arr[0];
		for (int i = 1; i<arr.length; i++){
			if(arr[i]>max){
				max=arr[i];
			}
		}
		return max;
	}

	public static void main(String[] args) {
		int array[] = IntArrayUtils.genRandom(7, -20, 20);
		String arrayToString = IntArrayUtils.toString(array);
		System.out.println("Масив " + arrayToString);
		System.out.println("Min: " + min(array));
		System.out.println("Max: " + max(array));
	}
}

Примерен резултат:
Масив {-4, -12, -15, 6, -20, -19, 3}
Min: -20
Max: 6
```

Алгоритми за сортиране

Сортирането подрежда елементите на списък във възходящ (нарастващ) или низходящ (намаляващ) ред. В следващите примери са реализирани различни алгоритми за сортиране върху масив от числа. Те могат да се прилагат и върху други линейни структури от данни, като напр. java.util.ArrayList. Алгоритмите могат да бъдат приложени не само върху числа, но и върху масиви от низове, а също и върху масиви от

всякакви други обекти, които могат да бъдат сравнявани по някакви критерии, напр., хора да се сортират по възраст, студенти по успех и т.н. Ако е необходимо по-сложни обекти да бъдат сортирани по няколко критерия, алгоритмите се прилагат върху части от списъците.

Някои от алгоритмите за сортиране се изпълняват по-бързо, ако масивът в някаква степен (случайно) е предварително подреден, а при други алгоритми броят на операциите е константен.

По-елементарни за реализация са алгоритмите **сортиране чрез вмъкване, сортиране чрез пряка селекция** и **метод на мехурчето**. При тях сложността в най-лошия случай, (когато елементите са подредени в обратен ред) и в средния случай (когато елементите са равномерно разбъркани), е $O(n^2)$.

Алгоритмите **бързо сортиране (quick sort)** и **сортиране чрез двоично дърво** са по-добри и имат сложност O(n*log(n)). **Бързото сортиране с избор на един главен елемент** в най-лошия случай има сложност $O(n^2)$. В класа java.util.Arrays е реализиран метод sort() за **бързо сортиране с два главни елемента** и в най-лошия случай неговата сложност е O(n*log(n)).

Алгоритъмът за **сортиране чрез сливане (merge sort)** предполага наличието на два или повече сортирани масива, които трябва да се обединяват. От версия 8 на Java класът java.util.Arrays предлага методи parallelSort за различни типове масиви, които реализират този алгоритъм. Методите първоначално разделят входния масив на няколко по-малки. Всеки от масивите се сортира самостоятелно чрез метода за бързо сортиране и накрая се сливат в общ сортиран масив.

Ще покажем „по-елементарните" за реализация методи за сортиране, за по-задълбочено вникване в логиката на създаване и разбиране на алгоритмите. Разбира се, в практиката е препоръчително да се използват стандартните методи sort (за по-малки масиви) и parallelSort (за по-големи масиви).

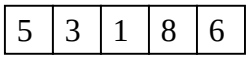
Сортиране чрез вмъкване

Идеята на алгоритъма за сортиране чрез вмъкване е следната:

1. Списъкът за сортиране се разделя се на две части: сортирана и несортирана;
2. При всяка стъпка се взема първият елемент от несортираната част и се вмъква на подходяща позиция в сортираната част. Това става като:
    a. Несортираният елемент се сравнява с всеки от елементите в сортирания списък;
    b. ако несортираният елемент е по-малък, той си разменя мястото с текущия елемент на масива, като по този начин несортираният елемент се придвижва напред;
    c. когато при сравнение несортираният елемент се окаже по-голям от текущо проверявания сортиран елемент, това означава, че той си е намерил правилната позиция.
3. Алгоритъмът приключва при изчерпване на списъка с несортираните елементи.

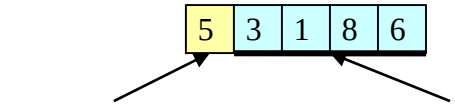
Така се извършва сортировка на елементите във възходящ ред. За да се сортират в низходящ ред, сравнението в точка 2.b трябва да е обратното – разменят се местата ако несортираният елемент е по-голям от текущо проверявания.

Нека разгледаме как се извършва сортировка във възходящ ред на примерен масив със стойности 5, 3, 1, 8, 6.



Начален списък

В началото (точка 1) може да приемем, че първият елемент е сортиран, а всички останали са в несортираната част.



Сортирана част Несортирана част

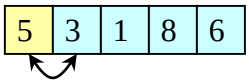
Втора част от алгоритъма (точка 2) се изпълнява в следния цикъл: взима се i-тия елемент, чиято стойност се изменя от 1 до последния индекс (в случая 4). За да се намери мястото на i-тия елемент измежду сортираните, се обхождат елементите пред него в друг вложен цикъл. Във вложения цикъл индексите, означени с j се изменят от i-тия елемент към нулевия. Сравнява се несортираният елемент (точка 2.а) с всеки j-ти сортиран елемент и ако несортираният има по-малка стойност, двата елемента си разменят местата. Във вътрешния цикъл не е необходимо да се обхождат всички сортирани елементи, а само докато се срещне елемент, който е по-малък от несортирания. Вътрешният цикъл приключва или с последната проверка (когато j стане равно на 0) или ако j-тия елемент е по-малък от несортирания елемент.

В следващия код е показана една възможна реализация на алгоритъма за сортиране чрез вмъкване – метод insertionSort. След извикване на метода, сортираният масив се получава във фактическия параметър, (а не като върнат резултат). Това е възможно, тъй като масивът се предава по адрес, чрез референция.

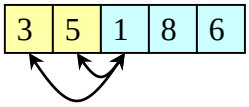
```
public class SortingAlgorithms {
    // Сортиране чрез вмъкване
    static void insertionSort(int [] arr) {
        // обхождаме всички несортирани елементи
        for (int i = 1; i < arr.length; i++) {
            // запомняме първия (i-ти) несортиран елемент
            int unsorted_element = arr[i];
            //ще използваме j след цикъла, затова я декларираме над него
            int j;
            // във вътрешния цикъл избутваме назад с по един
            // сортираните елементи, докато са по-големи от несортирания
            for (j = i; j > 0 && unsorted_element < arr[j-1]; j--) {
                arr[j] = arr[j-1];
            }
            // несортираният елемент поставяме на мястото на
            // последния по-голям от него сортиран елемент
            arr[j] = unsorted_element;
        }
    }

    public static void main(String[] args) {
            int [] array = {5, 3, 1, 8, 6};

            String arrayToString = IntArrayUtils.toString(array);
            System.out.println("Несортиран масив: " + arrayToString);

            insertionSort(array); // Сортиране чрез вмъкване
            arrayToString = IntArrayUtils.toString(array);
            System.out.println("Сортиран масив: " + arrayToString);
    }
}
```

```
Резултат:
Несортиран масив: {5, 3, 1, 8, 6}
Сортиран масив: {1, 3, 5, 6, 8}
```

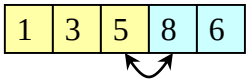
Нека покажем с картинки как се изпълнява сортирането върху примерния списък:



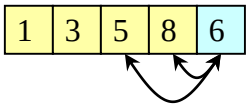
i=1, unsorted_element = 3
Във вложения цикъл се сравняват стойностите на първия и втория елемент на масива (5>3), и те разменят местата си.



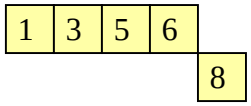
i=2, unsorted_element = 1
Във вложения цикъл се сравнява 1 с елементите 5 и 3 и тъй като са по-големи, се избутват назад, а несортираният се премества в началото.



i=3, unsorted_element = 8
Във вложения цикъл се сравнява 8 с елемента 5 и не се извършва размяна, защото 8>5.



i=4, unsorted_element = 6
Във вложения цикъл се сравнява 6 с елементите 8 и 5. Разменят се само само 6 и 8. Не се проверяват други елементи, защото 6 е по-голямо от 5.



Край

Недостатък на този метод е, че се извършват множество разместания на елементи. Предимство е, че вътрешният цикъл може да приключва бързо, без да се правят всички проверки. В най-добрия случай сложността на алгоритъма може да е O(n), но обикновено тя е $O(n^2)$.

Сортиране чрез пряка селекция

Логиката на сортирането чрез пряка селекция е следната:

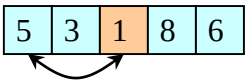
1. Списъкът за сортиране се разделя се на две части: сортирана и несортирана;
2. Намира се най-малкия елемент в несортираната част.
3. Разменят се местата на намерения минимален с първия елемент в несортираната част.
4. Първият елемент от несортираните става последен от сортираните, а несортираната част намалява с един елемент.
5. Алгоритъмът продължава с точка 2, докато свършат несортираните елементи.

За реализацията на този алгоритъм отново са необходими два цикъла: във външния цикъл последователно се взима i-тия елемент, като i се променя от 0 до предпоследния. Във вътрешен цикъл се намира индексът на минималния измежду елементите от i-тия до последния. След това се разменя намереният минимален с i-тия.

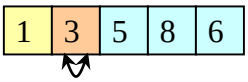
В следващия код е показан само методът за сортиране selectionSort, а използването му е подобно на вече разгледаното извикване на метода insertionSort.

```
    // Сортиране чрез пряка селекция
    static void selectionSort(int [] arr) {
          // Обхождаме всички елементи без последния.
          // Накрая последния ще си бъде на мястото.
          for (int i = 0; i < arr.length-1; i++) {
                // намираме индекса на минималния елемент
                // в частта от масива в интервал [i, arr.length]
                int indexOfMin = i;
                for (int j = i; j<arr.length; j++) {
                      if(arr[j]<arr[indexOfMin]){
                            indexOfMin = j;
                      }
                }
                // размяна на i-тия и минималния елемент
                int temp = arr[i];
                arr[i] = arr[indexOfMin];
                arr[indexOfMin] = temp;
          }
    }
```

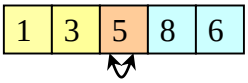
Нека разгледаме визуално как се изпълнява сортирането чрез пряка селекция върху вече разгледания примерен списък:



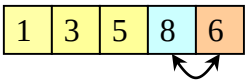
i=0
Във вложения цикъл намираме индекса на минималния елемент 1: indexOfMin→2. Разменяме го с нулевия (i=0).



i=1
Във вложения цикъл намираме индекса на минималния елемент 3: indexOfMin→1. Разменяме го с i-тия (i=1) т.е. остава си на мястото.



i=2
Във вложения цикъл намираме индекса на минималния елемент 5:indexOfMin→2. Разменяме го с i-тия (i=2) т.е. остава си на мястото.



i=3
Във вложения цикъл намираме индекса на минималния елемент 6: indexOfMin→4. Разменяме го с i-тия (i=3).

Край

Недостатък на този метод е, че при търсенето на минимален елемент се обхождат всички несортирани елементи. Така броят на извършваните операции винаги е еднакъв – $O(n^2)$, без значение дали списъкът е частично сортиран или не. Предимството му пред метода за сортиране чрез вмъкване е, че се извършват по-малък брой размени на елементи.

Метод на мехурчето

Идеята на метода на мехурчето се осъзнава най-лесно при вертикалното представяне на списъка (фиг. 8.1). Като се започне от края на списъка, изкачвайки се нагоре, се извършва сравняване на всеки два съседни елемента (първо последния с предпоследния,

после предпоследния с пред-предпоследния и т.н.) и ако е необходимо те си разменят местата така, че те да са сортирани. При първото обхождане най-лекият (най-малкият) елемент (мехурчето) ще изплува най-отгоре – на позиция 0. При всяко обхождане на списъка, действията (сравнения и размени) се извършват само върху неподредените елементи и най-лекият от тях изплува най-отгоре. Сортирането приключва, ако при някое от обхожданията не е направена нито една размяна.

Първо нека разгледаме реализация, при която не се прави проверка за размяна:

```
// Метод на мехурчето
static void bubbleSort(int [] arr) {
      // Обхождаме масива (arr.length-1) пъти
      for (int i = 1; i<arr.length; i++) {
            // Променяме j от последния до i-тия индекс със стъпка -1
            for (int j = arr.length-1; j>=i; j--) {
                  // Ако j-тия ел. е по-малък от предходния, ги разменяме
                  if(arr[j]<arr[j-1]){
                        int temp = arr[j];
                        arr[j] = arr[j-1];
                        arr[j-1] = temp;
                  }
            }
      }
}
```

За да се проверява за размяна във външния цикъл, се използва булева променлива, която се инициализира със стойност false. Ако във вътрешния цикъл се извърши промяна, стойността ѝ става true. Ако във външния цикъл не се е извършила промяна, изпълнението на програмата приключва. Тази логика е добавена в следващия код. Добавен е и код за трасиране, показващ състоянието на масива при всяка итерация на вътрешния цикъл.

```
static void bubbleSortTrace(int [] arr) {
      String arrayToString = IntArrayUtils.toString(arr);
      System.out.println("i,j - " + arrayToString);
      for (int i = 1; i<arr.length; i++) {
            boolean haveChange = false;
            for (int j = arr.length-1; j>=i; j--) {
                  if(arr[j]<arr[j-1]){
                        int temp = arr[j];
                        arr[j] = arr[j-1];
                        arr[j-1] = temp;
                        haveChange=true;
                  }
                  arrayToString = IntArrayUtils.toString(arr);
                  System.out.println(i + "," + j + " - " + arrayToString);
            }
            if(!haveChange){
                  break;
            }
      }
}
```

Резултат, в който показваме стойностите на i и j, и в който сме оцветили текущо сравняваните елементи, след евентуалната им размяна:

```
i,j - {5, 3, 1, 8, 6} – начален масив
1,4 - {5, 3, 1, 6, 8}
1,3 - {5, 3, 1, 6, 8}
```

```
1,2 - {5, 1, 3, 6, 8}
1,1 - {1, 5, 3, 6, 8}
2,4 - {1, 5, 3, 6, 8}
2,3 - {1, 5, 3, 6, 8}
2,2 - {1, 3, 5, 6, 8}
3,4 - {1, 3, 5, 6, 8}
3,3 - {1, 3, 5, 6, 8}
```

Двоично търсене

При търсене на стойност в подреден масив, не е нужно да се обхождат всичките му елементи. Търсената стойност се сравнява със средния елемент от масива. Ако двете стойности съвпадат, методът връща индекса на текущия елемент. Ако търсената стойност е по-малка от средния елемент, търсенето продължава само в частта с по-малките елементи. Иначе – в частта с по-големите елементи. Търсенето се повтаря върху смаляващите се части от масива, докато се намери елемент или не останат елементи за претърсване.

За организиране на търсенето, се използват три променливи за ляв, десен и среден индекс. Средният индекс се изчислява като средно-аритметично на левия и десния индекси. Ако средният елемент е по-голям търсения, търсенето продължава в лявата част на масива, като за целта десният индекс става равен на „средния индекс-1”. В противен случай левият индекс става равен на „средния индекс+1”. При всяка стъпка масивът, в който се извършва търсенето, намалява наполовина.

Примерна реализация е показана в следващия код. В основния – main метод се генерира масив със случайни елементи, сортира се и след това върху него се изпълнява търсенето.

```
public class BinarySearch {
  // Двоично търсене
  public static int binarySearch(int[] arr, int searchVal) {
    int left = 0;                       // ляв индекс
    int right = arr.length;             // десен индекс
    int middle = (left + right) / 2;    // среден индекс

    // В цикъла ще стесняваме интервала за търсене като
    // променяме трите променливи. Затова той се изпълнява
    // "докато левия индекс е по-малък или равен на десния"
    while (left <= right) {
      if (searchVal == arr[middle]) {     // ако сме намерили ел.
        return middle;                    // връщаме индекса му
      } else {                            // иначе
          if (searchVal < arr[middle]) { // ще търсим наляво
            right = middle - 1;
          } else {                        // или надясно
            left = left + 1;
          }
      }
      middle = (left + right) / 2;
    }
    return -1; // връщаме -1, ако не е намерен
  }

  public static void main(String[] args) {
    // генерираме случаен масив с 10 ел. в интервал [0,10]
    int array[] = IntArrayUtils.genRandom(10, 0, 10);
    // сортираме с метода QuickSort, реализиран в java.util.Arrays
    java.util.Arrays.sort(array);
```

```
    String arrayToString = IntArrayUtils.toString(array);
    int searchVal = 3; // търсена стойност
    int searchPos = binarySearch(array, searchVal); // намерена позиция
    System.out.println("В масива "+ arrayToString + " елемент " + searchVal
      +(searchPos==-1?" не е намерен":" е намерен на позиция "+searchPos));
  }
}

Примерни резултати:
В масива {0, 5, 6, 6, 6, 7, 8, 8, 9, 10} елемент 3 не е намерен
В масива {1, 1, 3, 3, 4, 6, 7, 7, 8, 9} елемент 3 е намерен на позиция 2
```

Вместо нашата потребителска реализация може да се използва стандартният метод за търсене java.util.Arrays.binarySearch(), който при намерен елемент връща индекса, а при ненамерен – число, по-малко от нула.

Трябва да напомним, че двоичното търсене се изпълнява правилно само при сортирани масиви. Прилагането му върху несортирани масиви не връща коректен резултат.

Въпроси и задачи за упражнения

1. Създайте масив от 10 елемента от тип double. Като стойности на всеки елемент, задайте стойности, равни на индекса на елемента. С отделни методи отпечатайте масива, покажете сумата от елементите му и произведението на ненулевите му четни елементи.
2. Въведете от конзолата масив с n елемента, които са реални числа. Отпечатайте го в обратен ред. Пресметнете средно-аритметичната стойност на числата, които са по-големи от нула.
3. За два масива от реални числа, проверете кой е с по-голяма сума на елементите.
4. За въведено от конзолата число n намерете факториелите на всички числа от 0 до n. Отпечатайте информация за намерените факториели и намерете сумата на факториелите от нечетните числа.
5. За въведени от конзолата естествени числа n и k (k<n), намерете по най-оптимален начин броя на всевъзможните комбинациите на k елемента измежду n елемента.
6. Редицата на Фибоначи е последователност от цели числа, при които първите два елемента имат стойности 0 и 1, а всеки следващ се образува като сума на предходните два. Отпечатайте първите 90 елемента от редицата. Отпечатайте като реални числа резултатите от делението на последните 2 елемента (предпоследния/последния и обратно). Какъв е резултатът?
7. Отпечатайте първите 100 елемента от редицата на Фибоначи.
8. Модифицирайте метода за двоично търсене така, че да извежда информация за броя на извършените сравнения за търсен елемент. Изпробвайте го върху случайно генериран голям масив, напр., с 3000 елемента в интервал [0, 2000] и търсени стойности 3 и 1000. Какви са изводите Ви?
9. Създайте нов метод, оптимизиращ описания в предходната задача метод за двоично търсене. Тествайте новосъздадения метод и метода от предходната задача върху едни и същи масиви.
10. Създайте метод за сортиране чрез сливане на два масива, който връща като резултат получения масив.
11. Реализирайте алгоритъма за бързо сортиране върху масив от цели числа.
12. Въведете от конзолата число N (N>5) и N на брой точки с координати (x, y) в R^2, като не се допуска две точки да съвпадат. Съхранявайте координатите за точките по x и по y в едномерни масиви. Намерете разстоянията между всеки две точки и ги

запишете в нов едномерен масив. Отпечатайте масивите. За всяка двойка точки намерете и отпечатайте сумата от разстоянията, средно-аритметичната им стойност, минималното и максималното разстояние.

Упътвания към задачите

1. Декларирайте едномерен масив. При обхождането му в цикъл задавайте стойностите на елементите. За създаването на методите използвайте примерите за цели числа.
2. Използвайте разгледаните в главата примери за работа с цели числа. За пресмятане на средно-аритметично на числата, които са по-големи от нула, използвайте показния метод за средно-аритметично, като добавите и брояч за елементите, отговарящи на условието.
3. Намерете поотделно сумите на двата масива и ги сравнете.
4. Може да се създаде масив от n+1 елемента, като първия има стойност 1. Стойността на всеки следващ елемент е равна на стойността на предходния елемент от масива, умножена по текущия индекс. Така в i-тия елемент на масива ще се съхранява i!.
5. Броят на комбинациите на n елемента от k-ти клас се описва с формулата C(n, k)=n!/(k!*(n-k)!). За да се намери отговорът по оптимален начин може да се използва предходната задача.
6. Създайте масив с елементи от тип long. Използвайте логиката от предходната задача.
7. Използвайте логиката от предходната задача. Стотният елемент надхвърля границите на типа long. Декларирайте като тип на елементите на масива BigInteger или BigDecimal.
8. В метода за двоично търсене добавете променлива-брояч на операциите. Не забравяйте да сортирате масива преди да приложите метода за двоично търсене.
9. Възможна оптимизация е „средният елемент” да се избира не като средно-аритметично на индексите, а чрез предполагаемия индекс за местоположението на търсеното число. Ако масива е означен с arr и търсеното число – с x, формулата за предполагаем индекс може да е middle = left+(x-arr[left])*step, където step=(right-left)/(arr[right]- arr[left]). Защо?
10. Ако броят на елементите на масивите, които ще се сливат, е n и m, се създава нов масив с length=(n+m) елемента. Задават се три управляващи променливи i, n, m, по една за всеки един от трите масива, с начални стойности 0. В един цикъл, при обхождане по i-тия елемент на новия масив се сравняват n-тия и m-тия елементи от двата сортирани масива, по-малкият се добавя на i-то място и стойността на съответния му указател – n или m, се увеличава с единица.
11. Потърсете обяснения и примерни реализации в Интернет.
12. Броят на разстоянията между N точки може да се пресметне като сума на аритметична прогресия N*(N-1)/2 или като комбинация от N елемента втори клас, което се записва като C(N, 2)=N!/(2!*(N-2)!). Разстоянията се изчисляват по питагоровата теорема. За пресмятане на разстоянията между всички двойки точки, се използват два вложени цикъла. Във външния цикъл се обхождат точките по i от 0 до N-1. Във вложения се обхождат точките по j от i+1 до N. Във вложения цикъл се увеличава и индекса на текущо изчислявания елемент от масива с разстоянията. Аналогично, с двата вложени цикъла може да се извършват и другите необходими действия върху трите масива, както и да се следи за съответствията между точки и конкретни разстояния.

Глава 9. Многомерни масиви

Двумерни, тримерни и т.н. n-мерни масиви се наричат с общото име многомерни.

Двумерните масиви може да се разглеждат като матрици, като таблици с редове и колони, а също и като едномерни масиви, всеки от елементите на които също е едномерен масив. При тях достъпът до всеки елемент се осъществява чрез два индекса – за ред и колона, като всеки от индексите се изменя от 0 до „броя на редовете/колоните – 1”.

индекси i \ **j**	**0**	**1**	**2**	**3**
0	[0][0]	[0][1]	[0][2]	[0][3]
1	[1][0]	[1][1]	[1][2]	[1][3]
2	[2][0]	[2][1]	[2][2]	[2][3]

стойности i \ **j**	**0**	**1**	**2**	**3**
0	3	4	3	2
1	8	7	9	0
2	0	4	5	4

В линейната алгебра и аналитичната геометрия е разработен специализиран математически апарат за работа с матрици и вектори, (които се представят чрез единични масиви). Матрица с m реда и n колони се представя като таблица, в която всеки ред има точно по n елемента.

Матриците имат приложения в множество различни области на математиката и на други науки: за автоматизирано решаване на системи линейни и диференциални уравнения с множество неизвестни; в компютърната графика за трансформации и транслации на тримерни обекти в двумерното пространство, (каквото е компютърният екран); в икономиката, математиката и информатиката за решаването на оптимизационни задачи по транспортиране, производство и др., използващи симплекс метода, за намиране на оптимално разпределение на ресурсите с цел минимизиране на разходите и максимизиране на печалбите при определени ограничения; във физиката, химията, биологията за представяне на различни обекти и др.

В реалния свят не винаги двумерните масиви е необходимо да имат равен брой елементи по редове и колони. Напр., групи от m измервания, всяка от които има различен брой измервания могат да бъдат представени в двумерен масив с m реда, но различен брой елементи в него т.е. в този случай може да гледаме на двумерния масив като на едномерен масив от едномерни масиви. Удобството от представянето по този начин би могло да бъде в това, че могат да бъдат записани в една структура и обработвани по еднотипен начин.

индекси i \ **j**	**0**	**1**	**2**	**3**
0	[0][0]	[0][1]	[0][2]	
1	[1][0]	[1][1]		
2	[2][0]	[2][1]	[2][2]	[2][3]

стойности i \ **j**	**0**	**1**	**2**	**3**
0	3	4	3	
1	8	7		
2	0	4	5	4

За многомерните масиви може да се мисли като многомерно пространство, за което в конкретните точки за координатите се задават стойности на някаква функция. Например, в точка с координати (a_i, $a_{j,\ldots,}a_k$) се записва стойността $x_{ij\ldots k}$. Тензорите в математиката, използвани и във физиката се представят като многомерни масиви. Частен случай на тензорите всъщност, са скаларите (или числата), които са 0-мерни стойности, векторите – едномерни масиви от числа и матриците – двумерни масиви от числа. Докато при асоциативните масиви, поддържани в други езици за програмиране, ключовете могат

да бъдат с произволни стойности и типове, използването на обикновени масиви в Java ограничава свободата при задаване на произволни координати в дискретните координатни системи на евентуално различните измерения. За по-сложни структури от данни може да се използват и възможностите на обектно-ориентираното програмиране.

В тази част ще представим различни начини за създаване на двумерни масиви и някои основни методи за работа с тях.

Инициализация на двумерни масиви

Подобно на едномерните масиви, при декларацията на двумерни масиви може да се задават конкретни стойности на елементите им. По следния начин се декларира двумерен масив с 2 реда и 3 колони, елементите на които са цели числа от тип int.

```
int[][] matrix = {
            { 1, 2, 3 },
            { 4, 5, 6 }
       };
```

Във фигурни скоби, за всеки отделен ред се задава списък от стойностите за съответните колони. За разделител между описанията на редовете се използва запетая. За удобство при четенето, отделните редове на масива се описват в отделни редове на кода.

По подобен начин се задават и двумерни масиви, които имат различен брой елементи в отделните редове:

```
int[][] array2D = {
            { 1, 2, 3 },         // нулевия ред има 3 елемента
            { 4, 5, 6, 7, 8 },   // ред с индекс 1 има 5 елемента
            { 9, 0 }             // ред с индекс 2 има 2 елемента
       };
```

Достъп до елементите на двумерни масиви

Елементите на двумерните масиви се достъпват с името на променливата и два индекса – за ред и колона, зададени в правоъгълни скоби. С първия индекс се определя реда, а с втория – колоната. И двата индекса имат начални стойности 0 (нула). Например, за достъп до елементите от първия ред на масива array2D, първия индекс е 0, а за колоните – 0, 1, 2. Тогава, към елементите се обръщаме по следния начин: array2D[0][0], array2D[0][1], array2D[0][2]. За елементите от следващия ред, първия индекс е 1, а индекса за колоните отново се увеличава от 0 със стъпка 1: array2D[1][0], array2D[1][1], array2D[1][2], array2D[1][1], array2D[1][4] и т.н.

Заделяне на памет за двумерни масиви

С оператора new може да се заделя памет за двумерните масиви динамично, по време на изпълнение на програмата. Например, за матрица с 2 реда и 3 колони, с елементи от тип int, памет се заделя така:

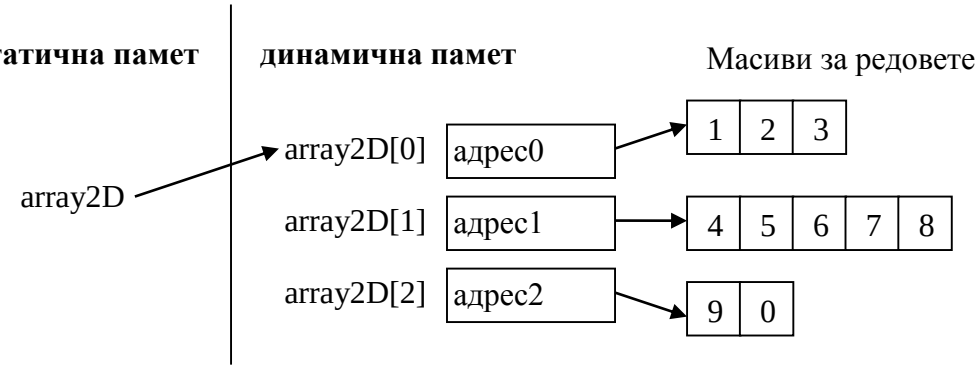
```
int rows = 2;  // брой редове
int cols = 3;  // брой колони
int[][] matrix = new int[rows][cols];
```

След оператора new се записва типа на елементите на матрицата, в първите правоъгълни скоби се задава броя на редовете на матрицата, в следващите – броя на колоните.

Ако желаем броят на елементите за всеки ред да е различен, първоначално с оператора new, трябва да се създаде само едномерен масив с референции за редовете. Това става като се зададе стойност за редовете и не се задава стойност за колоните, напр.,

```
int rows = 3;  // брой редове
int [][] array2D = new int[rows][];
```

Променливата за двумерния масив е референция към едномерен масив. В него се записват референции към едномерни масиви, съдържащи стойностите по редове.



Фиг. 9.1. Двумерни масиви в паметта

След това (за случая, при който броят на елементите по редове е различен), за всеки от масивите с елементи по редове се заделя памет самостоятелно:

```
array2D[0] = new int[3]; // в ред с индекс 0 има 3 елемента
array2D[1] = new int[5]; // в ред с индекс 1 има 5 елемента
array2D[2] = new int[2]; // в ред с индекс 2 има 2 елемента
```

Брой на редовете и на колоните в двумерен масив

Броят на редовете и колоните, както и при едномерните масиви, може да се вземе автоматично чрез полето length. Приложено върху променливата за масив чрез length получаваме броя на редовете, а върху всеки отделен ред – броя на елементите (колоните) в конкретния ред:

```
matrix.length      // брой редове на matrix
array2D.length     // брой редове на array2D
matrix[0].length   // брой елементи(колони) в ред 0 на matrix
matrix[i].length   // брой елементи(колони) в ред i на matrix
array2D[i].length  // брой елементи(колони) в ред i на array2D
```

Обхождане на двумерен масив

За да се премине през всички възможни комбинации от индекси на двумерен масив, се използват два цикъла, с две управляващи променливи i и j. В единия, външен цикъл, се изменя индекса по редовете i със стъпка 1, от 0 до „броя на редовете-1”. Във вложения цикъл, за всеки от редовете, се променя j със стъпка 1, от 0 до „броя на колоните за текущия ред-1”. Напр.,

```
for (int i = 0; i < matrix.length; i++) {
      <начални действия върху ред matrix[i]>
      for (int j = 0; j < matrix[i].length; j++) {
            <действия върху елемент matrix[i][j]>
      }
      <заключителни действия върху ред matrix[i]>
}
```

По този начин **елементите на двумерния масив се обхождат последователно отляво-надясно и отгоре-надолу.** Аналогично, като се смени начинът на промяна на i и j, може да реализира обхождане на елементите на двумерния масив и по други 3 начина: отляво-надясно и отдолу-нагоре; отдясно-наляво и отгоре-надолу; отдясно-наляво и отдолу-нагоре.

Въвеждане на матрица от конзолата

В следващия пример от конзолата се въвеждат две числа – rows и cols, съответно за брой на редовете и колоните на матрица. След това, също от конзолата се въвеждат всички елементи на матрица с елементи от тип int.

```
import java.util.Scanner;

public class ConsoleMatrix {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);

            System.out.print("Въведете брой на редовете на матрицата:");
            int rows = scanner.nextInt();

            System.out.print("Въведете брой на колоните на матрицата:");
            int cols = scanner.nextInt();

            int [][] matrix = new int[rows][cols];
            System.out.println("Въведете елементите на масива:");
            for (int i = 0; i < matrix.length; i++) {
                  for (int j = 0; j < matrix[i].length; j++) {
                        System.out.print("a[" + i + "][" + j + "]=");
                        matrix[i][j] = scanner.nextInt();
                  }
            }

            scanner.close();
      }
}
```

Примерно използване:

```
Въведете брой на редовете на матрицата:2
Въведете брой на колоните на матрицата:3
Въведете елементите на масива:
a[0][0]=1
a[0][1]=2
a[0][2]=3
a[1][0]=4
a[1][1]=5
a[1][2]=6
```

Отпечатване на двумерен масив в конзолата

За отпечатване на двумерен масив създаваме метод, който получава двумерен масив като параметър и връща масива като низ. С параметъра от булев тип inline се указва, дали резултатния низ да съдържа символи за нов ред след отпечатването на ред от масива. В main метода са показани и двата варианта за извеждане на двумерен масив.

```
public class IntArray2DUtils {
  // Метод, връщащ като низ двумерния масив, получен като параметър.
  // Низът съдържа символи за нов ред, ако inline е false, и не
  // съдържа - ако е true.
  public static String toString(int[][] array2D, boolean inline) {
   // Низ, който добавяме за край на ред
   String newLine = inline ? "" : "\n";
   String str = "{" + newLine;

   for (int i = 0; i < array2D.length; i++) {
      str += " {";
      for(int j=0; j < array2D[i].length; j++){
        str += array2D[i][j] + (j < array2D[i].length - 1 ? ", " : "");
      }
      str += "}" + (i < array2D.length - 1 ? "," : "") +  newLine;
   }

   str += "}";

   return str;
  }

  public static void main(String[] args) {
    int[][] array2D = {
      {1, 2, 3},
      {4, 5, 6, 7, 8},
      {9, 0},
    };

    // масива в един ред
    String str = toString(array2D, true);
    System.out.println("Масива на един ред: " + str);

    // масива в няколко реда
    str = toString(array2D, false);
    System.out.println("Масива на няколко реда:\n" +str);
  }
}

Резултат:
Масива на един ред: { {1, 2, 3}, {4, 5, 6, 7, 8}, {9, 0}}
Масива на няколко реда:
{
 {1, 2, 3},
 {4, 5, 6, 7, 8},
 {9, 0}
}
```

Подобно, но по-силно действие има метода deepToString на класа java.util.Arrays, който връща като низ не само двумерни, но и масиви с по-голяма размерност.

```
int [][][] arr = {
            {
                  {1,2,3},
                  {4,5,6}
            },
            {
                  {7, 8}
            }
      };
String str = java.util.Arrays.deepToString(arr);
System.out.println(str);

Резултат:
[[[1, 2, 3], [4, 5, 6]], [[7, 8]]]
```

Генериране на двумерен масив от случайни числа

За генериране на двумерен масив от случайни числа се използват класовете java.util.Random и java.util.stream.IntStream. Към вече създадения клас за отпечатване е добавен и методът int[][] genRandom(int rows, int cols, int n, int m, boolean isMatrix), който е използван в следващи примери. Методът връща двумерен масив с rows реда и до cols колони; елементите му са случайни числа в интервала [n, m]; и ако булевият параметър isMatrix има стойност true, броят на колоните във всички редове е еднакъв, иначе броят е случайно число в интервала [1, cols].

Особеност в метода е, че в цикъл създаваме самостоятелно за всеки ред едномерен масив от случайни числа и го присвояваме като стойност на i–тия ред intArray2D[i] на двумерния масив. Това е възможно, тъй като intArray2D[i] е референция, която може да сочи към различни адреси.

```
import java.util.Random;
import java.util.stream.IntStream;

public class IntArray2DUtils {
  // Методът генерира двумерен масив с rows реда и cols колони
  // с елементи от тип int в интервал [n, m].
  // Ако isMatrix е true, всички редове имат равен брой колони - cols,
  // иначе броят на колоните за всеки ред е случайно число от 1 до cols.
  public static int[][] genRandom(int rows, int cols,
                                  int n, int m, boolean isMatrix) {
    Random rand = new Random();
    int[][] intArray2D = new int[rows][];  // масив за редовете
    for (int i = 0; i < intArray2D.length; i++) {  // за всеки ред
      // задаваме броя на елементите на текущия ред
      int currentCols = isMatrix ? cols : rand.nextInt(cols) + 1;
      // генерираме случаен поток с currentCols елемента
      IntStream is = rand.ints(currentCols, n, m + 1);
      // на i-тия ред присвояваме съответния на потока едномерен масив
      intArray2D[i] = is.toArray();
    }
    return intArray2D;  // връщаме двумерния масив
  }

  public static String toString(int[][] array2D, boolean inline) {
      ...
  }

  public static void main(String[] args) {
    // Генериране на матрица от 2 реда и 3 колони с ел. в инт. [0, 20]
    int[][] array2D = genRandom(2, 3, 0, 20, true);
    String str = toString(array2D, true);
    System.out.println(str);
```

```
    // Генериране на двумерен масив с 4 реда и до 5 колони с ел. в инт. [0, 20]
    array2D = genRandom(4, 5, 0, 20, false);
    str = toString(array2D, true);
    System.out.println(str);
  }
}

Примерен резултат:
{ {1, 4, 13}, {19, 3, 12}}
{ {17}, {15, 11, 10}, {15, 17}, {13, 3, 14, 2, 5}}
```

Основни понятия и действия с матрици

В линейната алгебра има основни понятия и основни действия върху матрици. Ще ги опишем накратко.

Матрица с равен брой редове и колони се нарича квадратна матрица и се означава M_{N*N}. **Основни понятия за квадратни матрици:**

- ***главен диагонал*** – множеството от елементи M[i][i], за които индексите за ред и колона са равни;
- ***втори (вторичен) диагонал*** – множеството от елементите M[i][N-i-1];
- ***горно-триъгълна матриц***а – елементите под главния диагонал са равни на 0;
- ***долно-триъгълна матрица*** – елементите над главния диагонал са равни на 0;
- ***диагонална матрица*** – елементите в страни от главния диагонал са равни на 0;
- ***скаларна матрица*** – диагонална матрица, за която елементите от главния диагонал имат равни стойности;
- ***единична матрица*** – скаларна матрица, за която елементите от главния диагонал са равни на 1;
- ***нулева матрица*** – матрица, за която всички елементи са равни на 0, и др.

Основни операции с матрици:

- ***събиране на еднотипни матрици A_{N*M} и B_{N*M}*** с по N реда и M колони – резултатът е матрица $C_{N*M}= A_{N*M}+B_{N*M}$, всеки елемент на която е сума от съответните елементи на матриците A и B;
- ***умножение на матрица A_{N*M} с число x*** – резултатът е матрица $B_{N*M}=x*A_{N*M}$, всеки елемент на която е произведение на числото със съответния елемент на матрицата A;
- ***умножение на матрица A_{N*M} с вектор B_M*** с M елемента – резултатът е вектор $C_N=A_{N*M}*B_M$, за който всеки елемент C[i] се получава като сумата от произведенията A[i][j]*B[j], където j се изменя от 0 до M-1;
- ***умножение на матрица A_{N*M} с матрица B_{M*K}*** – резултатът е матрица $C_{N*K}=A_{N*M}*B_{M*K}$, за който всеки елемент C[i][j] се получава като сумата от произведенията A[i][s]*B[s][j], където s се изменя от 0 до M-1, известно като правилото „ред по стълб”;
- ***противоположна на матрица A_{N*M}*** – резултатът е матрица $B_{N*M}=-A_{N*M}$, всеки елемент на която е равен на съответния елемент на матрицата A, но с обратен знак;
- ***намиране на транспонирана матрица, детерминанта, норма на матрица и др.***

Следват някои по-елементарни методи, свързани с основните понятия и действия с матрици.

Умножение на матрица с число

Умножението на матрица с число връща като резултат нова матрица, елементите на която са равни на съответните елементи на началната матрица, умножени по числото. За целта трябва динамично да се задели памет по редове и колони за новата матрица. След

това едновременно се обхождат всички елементи на началната матрица и новата с два цикъла и извършваме съответните умножения. В main метода използваме методите от класа IntArray2DUtils за генериране на случайна матрица и преобразуването ѝ в низ, както и новосъздадения метод за умножение на матрица с число.

```
public class MultiplyByNum{
  // Умножава цяло число с матрица от цели числа.
  // Резултатът е нова матрица.
  // Матрицата, зададена като параметър не се променя
  public static int[][] multiplyByNum(int[][] matrix, int num) {
    // заделяне на памет за референциите за редове
    int[][] resultMatrix = new int[matrix.length][];

    for (int i = 0; i < matrix.length; i++) {
      // заделяне на памет за текущия ред
      resultMatrix[i] = new int[matrix[i].length];
      for (int j = 0; j < matrix[i].length; j++) {
        resultMatrix[i][j] = matrix[i][j]*num; // умножение
      }
    }
    return resultMatrix;
  }

  public static void main(String[] args) {
    // матрица 4*5
    int[][] matrix = IntArray2DUtils.genRandom(4, 5, 0, 4, true);
    int num = 2;
    // нова матрица за резултата
    int[][] newMatrix = multiplyByNum(matrix, num);

    String strMatrix = IntArray2DUtils.toString(matrix, true);
    String strNewMatrix = IntArray2DUtils.toString(newMatrix, true);
    System.out.println(num + "*" + strMatrix + "\n= " + strNewMatrix);
  }
}

Примерен резултат:
2*{ {1, 3, 0, 4, 1}, {2, 3, 0, 1, 2}, {1, 0, 2, 3, 2}, {1, 1, 2, 4, 2}}
= { {2, 6, 0, 8, 2}, {4, 6, 0, 2, 4}, {2, 0, 4, 6, 4}, {2, 2, 4, 8, 4}}
```

Диагонали на квадратна матрица

За обхождане на главния или вторичния диагонал на квадратна матрица с n реда и n колони, се използва само един цикъл, тъй като за диагоналите има зависимост между индексите по редове и колони. Елементите по главния диагонал са [i][i], а по вторичния [i][n-i-1], където i се изменя от 0 до n-1.

В следващия пример са създадени методи за намиране на сума по главния диагонал и минимален елемент по вторичния. В методите не е направена проверка за коректност на входната матрица, която трябва да има равен брой редове и колони. Читателят може самостоятелно да я добави, като предварително обходи всички редове и сравни дължината им (т.е. колоните) с броя на редовете.

```
public class DiagonalMatrix {
  // Връща сумата на елементите по главния диагонал
  public static int sumByMainDiagonal(int[][] matrix) {
    int sum = 0;
    for (int i = 0; i < matrix.length; i++) {
      sum += matrix[i][i];
```

```
    }
    return sum;
  }

  // Връща минималния от елементите по вторичния диагонал
  public static int minBySecondaryDiagonal(int[][] matrix) {
    int n = matrix.length;
    int min = matrix[0][n-1];
    for (int i = 1; i < n; i++) {
      if(matrix[i][n-i-1]<min){
        min = matrix[i][n-i-1];
      }
    }
    return min;
  }

  public static void main(String[] args) {
    // матрица 4*4
    int[][] matrix = IntArray2DUtils.genRandom(4, 4, 0, 9, true);
    int sum = sumByMainDiagonal(matrix);
    int min = minBySecondaryDiagonal(matrix);

    String strMatrix = IntArray2DUtils.toString(matrix, false);
    System.out.println("Матрица: " + strMatrix);

    System.out.println("Сума по главния диагонал: " + sum);
    System.out.println("Минимален елемент по вторичния диагонал: " + min);
  }
}
```

Примерен резултат:

```
Матрица: {
 {9, 7, 4, 8},
 {0, 4, 2, 4},
 {8, 8, 1, 0},
 {5, 7, 7, 0}
}
Сума по главния диагонал: 14
Минимален елемент по вторичния диагонал: 2
```

Въпроси и задачи за упражнения

1. Създайте клас за работа с матрици, чиито елементи са реални числа. Като отделни методи в него реализирайте проверки дали матрица е горно-триъгълна, долно-триъгълна, диагонална, скаларна, единична или нулева. Създайте методи за основните операции - намиране на обратна матрица, събиране на матрици, умножение на матрица с число, с вектор и с друга матрица от подходящ тип.
2. За матрици A_{N*M}, B_{N*M}, C_{N*M} намерете резултата от изчислението на израза $3*(A_{N*M}+B_{N*M})-C_{N*M}$.
3. В двумерен масив се записват данните от измерванията на температурата в няколко населени места. Всеки ред от масива описва температурите за един град, като при това не се указват допълнителни характеристики за ден, час и др. Създайте масив и го инициализирайте със случайни реални числа в интервала [-20, 45]. За всяко населено място намерете минималните и максималните температури, средните температури и средните положителни температури. Намерете максималните и минималните температури от средните температури по градове.
4. Шахматна дъска се представя като матрица с 8 реда и 8 колони. Разположете в 6 случайни клетки 6 различни бели фигури, определени чрез числа и символи по

следния начин: 1(Ц) – цар, 2(Д) – дама, 3(Т) – топ, 4(О) – офицер, 5(К) – кон, 6(П) – пешка. За всяка от фигурите определете възможните ходове и ги представете като координати от шахматната дъска – A1, B3 и т.н.

5. В една област има n града. Между някои от тях съществуват директни пътища. Определете двойките градове, за които не съществуват директни пътища, но може да бъде направен маршрут с преминаване през само един друг град.

Упътвания към задачите

1. За да разберете смисъла на понятията и действията с матрици, използвайте кратките определения в частта Основни понятия и действия с матрици, както и информация от Интернет. За всеки от методите, определете зависимости между индексите на елементите, които трябва да бъдат проверявани или обработвани. При необходимост нарисувайте примерни матрица с означени елементи и съответните им индекси.
2. Използвайте част от реализираните в предходната задача методи.
3. Запишете имената на n на брой населени места в едномерен масив от низове. Генерирайте двумерен масив от реални числа с n реда, при което всеки ред ще се отнася за населеното място, което има същия индекс от едномерния масив с населени места. Използвайте метода IntArray2DUtils.genRandom или създайте собствен метод за работа с реални числа. Създайте методи, чрез които може да изчислите минималната, максималната и средно-аритметичната стойност, както и средно-аритметичната на положителните стойности в едномерен масив. Приложете ги за всеки от редовете с температури и запишете резултатите в съответни масиви. Използвайте методите за минимален и максимален елемент и върху масива със средни температури.
4. Създайте матрица с размерност 8*8 от цели числа и я инициализирайте с нули. За всяка от шестте фигури генерирайте по два случайни индекса за ред и колона, като комбинациите не трябва да се повтарят. В генерираните позиции запишете съответните на фигурите букви. За всяка от фигурите, намерете възможните ходове, като се съобразите с правилата на играта Шах. Внимавайте фигура да не излиза от дъската и да не се получават колизии, (т.е. да не се преминава през други фигури). Например, на следващата картинка са дадени случайни разположения на фигурите и са оцветени възможните ходове на топа и пешката.

		A	B	C	D	E	F	J	H
		0	1	2	3	4	5	6	7
1	0				т				
2	1			Ц	т				
3	2	т	т	т	Т	т	т	Д	
4	3				т				
5	4		О		т		П		
6	5				т		п		
7	6			К	т				
8	7				т				

Начините на придвижване определят как се изменят индексите i и j спрямо текущата позиция. За различните фигури трябва да се изпълнят по няколко различни цикъла, за да се определят възможните ходове. Например топът може да се придвижва наляво,

надясно, нагоре и надолу спрямо текущата си позиция т.е. трябва да се реализира обхождане за четирите посоки чрез четири обикновени (не вложени) цикъла – за i--, i++, j--, j++.

5. Един начин за решение е следният. Създава се нулева матрица A_{n*n}, в която по редове и колони са означени градовете. Само за градовете, между които има директна връзка, в съответната клетка се задават единици. Тъй като връзките между градовете са ненасочени, матрицата ще е симетрична (в други задачи може връзките да са насочени и матрицата да не е симетрична). Втората степен на матрицата A (т.е. произведението A*A) ни дава информация за градовете, които имат връзка през друг град. По-точно, клетките на A^2, в които има единици показват, че между съответните градове има връзка през един друг град. В примера за „София-София", „София-Бургас", „Бургас-Бургас" има единици т.е. има връзка през друг град – и в трите случая през Пловдив. За Пловдив има два различни пътя – през Бургас и през София – и затова за „Пловдив-Пловдив" в матрицата се е получила стойност 2. Като премахнем излишните (с еднакви начало и край) връзки, ще остане търсеният резултат: „София-Бургас".

		0	1	2	...
		София	Пловдив	Бургас	...
0	София	0	1	0	
1	Пловдив	1	0	1	
2	Бургас	0	1	0	
...	...				

0	1	0	*	0	1	0	=	1	0	1
1	0	1		1	0	1		0	2	0
0	1	0		0	1	0		1	0	1

С третата степен A^3 може да се намери между кои градове има връзка през други два града и т.н.

Логиката, по която се получава информацията за връзките между градовете, е следната: i-тия ред (или колона) показва с кои градове има връзка i-тият град; умножението на i-тият ред с j-тата колона, е сума от произведенията на съответните (с равни индекси по ред и колона) елементи; всяко от произведенията показва дали i-тият град има връзка с j-тия град през град с индекс k; т.е. ако i-тият и j-тият град имат връзка с един и същи град, съответните k-ти индекси ще имат единици и тогава произведението им ще има стойност единица. В примера за оцветените в червено елементи, „връзките на София"*„връзките на Бургас" (ред с индекс 0 по колона с индекс 2), имат общ елемент „Пловдив" (с индекс 1). Тогава сумата от произведенията на елементите дава 1 (заради Пловдив), което се записва като стойност на елемента [0][2] в новата матрица.

Глава 10. Методи

В настоящата глава е представена детайлна информация за методите. Показан е синтаксисът за създаване на методи и смисълът на различните модификатори. Разгледани са различни начини за използване на методите и механизмите за предаване на параметри към тях.

Смисъл на методите

Методите са подпрограми, които могат да бъдат използвани многократно. В разглежданите задачи има много примери за многократно използване: върху различни числа може да се прилагат едни и същи математически функции; за различни масиви от входни данни, без значение от тяхното съдържание, може да се изпълняват едни същи операции – сортиране, намиране на максимален и минимален елемент и много други.

Методите решават относително самостоятелни задачи. Има методи, които реализират стандартни алгоритми и други, които решават специфични за конкретно задание проблеми. Множество сложни задачи могат да бъдат решени, като се сведат до други вече решени задачи или комбинация от решени задачи. Ако например, се търси път между населени места, задачата може да се сведе до умножение на матрици; ако се търси минимален елемент в масив, масивът може първо да се сортира и тогава търсеният резултат ще се намира в първия му елемент. Разбира се, при използване на готови методи трябва да се внимава да не се получи неефективно решение, което е бавно или използва много памет за съхранение на данни. Търсенето на минимален елемент в масив чрез сортирането му е удачно, само ако сортираният масив се използва и за други дейности – например за многократно двоично търсене. Това е така, защото сложността на търсенето на минимален елемент е линейна O(n), а сложността на сортирането е квадратична – O(n*log(n)) или $O(n^2)$.

Не е необходимо да се познава точната реализация на методите, за да се използват. Може да се работи с тях като с черни кутии: достатъчно е да се знае какво вършат, какви входни данни получават и какъв изходен резултат връщат. В повечето потребителски програми например, се използва методът System.out.println (или print), който отпечатва текст в конзолата, без да е необходимо да се познава и разбира сорс кода му.

В процедурно-ориентираната логика, методите обикновено обработват входни параметри. Понякога използват глобални променливи и константи.

При обектно-ориентираната логика методите работят не само с входните параметри, но и с полетата на класа. Полетата са част от вътрешното състояние на обектите на класа, а методите са въздействия, които външната среда може да оказва върху обекта. Конкретните характеристики на външното въздействие, което променя вътрешното състояние на обект (т.е. стойностите на полетата му), се задават чрез параметрите на методите.

Въпреки че Java е обектно-ориентиран език за програмиране, в разглежданите досега примери сме използвали най-вече процедурно-ориентирана логика. При това не е било необходимо създаване на обекти на клас, за да се изпълняват методите, декларирани като статични.

Методите може формално да се разделят на процедури, които не връщат стойност и функции – които връщат. Тези понятия се използват в някои езици за програмиране.

Декларация на метод

Синтаксисът за декларация на метод е:

```
   [<модификатори>] [<тип на връщана стойност>] <име на метод>([<списък с
формални параметри>])[throws <списък с изключения>]
```

Модификаторите за метод са два вида и са незадължителни:

- ***модификатори за достъп* – *public, private, protected, по подразбиране*** (ако не е зададен) – определят различни видове видимост в класовете и пакетите;
- ***други модификатори* – *abstract, final, native, static, synchronized, strictfp*.**

Типът на връщана стойност на метод може да е:

- ***произволен тип*** – както примитивен, така и сложен. В този случай в тялото на метода трябва да се връща стойност от съответния тип с помощта на оператор return;
- ***void*** – т.е. методът не връща стойност;
- ***без указана връщана стойност*** – при методите-конструктори не се указва тип на връщана стойност;

Името на метод *отговаря на правилата за задаване на идентификатори*.

В **списъка с формални параметри** се описват ***променливи от произволни типове, разделени със запетаи*.**

Ако в метод възникват задължителни за обработка изключения, те могат да бъдат обработени в try-catch блок или да се укаже, че изключенията се предават към извикващия го метод. За целта е необходимо те да се опишат в **списъка с изключения**. Той съдържа имена на класове, разделени със запетаи.

Модификатори за достъп

Модификаторите за достъп определят нивото за видимост/достъп до методите на един клас от други класове (от същия или други пакети). Те се използват и за други членове на класа: полета и вложени класове. **Нивата за достъп са**:

- ***в методи на текущия клас;***
- ***в класове на текущия пакет;***
- ***в наследник на класа;***
- ***навсякъде.***

Следващата таблица показва кой модификатор какви нива на достъп задава:

модификатор/ ниво за достъп	текущ клас	текущ пакет	наследник на класа	навсякъде (друг пакет)
public	да	Да	да	да
protected	да	Да	да	не
(не е зададен)	да	Да	не	не
private	да	Не	не	не

Членовете на клас, които са описани с модификатор:

- **public,** са достъпни навсякъде;
- **protected** – достъпни са от всички класове в текущия пакет и от наследниците му, които може да не са в текущия пакет;
- **без модификатор** – достъпни са от всички класове, но само в текущия пакет;
- **private** – достъпни са само в текущия клас и не могат да се ползват извън класа.

Модификаторите за достъп са част от обектно-ориентираната концепция за програмиране, поради това няма да бъдат разгледани подробно в тази глава.

Тип на връщана стойност

Ако метод се използва само за извеждане на информация, обикновено, указваме че не връща стойност. За целта се използва ключовата дума void като тип на връщана стойност. Напр., основният main метод използва други методи и обикновено, за да се види какво е извършил отпечатва текст в конзолата; метод, който отпечатва минималното от две числа в конзолата, може да не връща минималното число:

```
public static void main(String[] args)
public static void showMin(int i1, int i2)
```

За да се използва метод като функция, трябва да се укаже от какъв тип са връщаните стойности. Напр., може да създадем метод min, който връща по-малкото от две цели числа, зададени като параметър; метод minByAge – връща човека от клас Person, който е с най-малка възраст измежду двама човека или в списък от хора; метод genRandom – генерира масив от случайни цели числа.

```
public static int min(int i1, int i2)
public static Person minByAge(Person p1, Person p2)
public static Person minByAge(Person [] persons)
public static int[] genRandom(int length, int n, int m)
```

Функциите може да участват в сложни изрази, в които се работи с величини от съответния тип. Напр.:

```
int a = 3 + min(5, 7)*2;
int b = 7;
int c = a + min(a, min(b, 4));
```

За да се върне резултат от изпълнението на метод, в тялото му се използва оператор return. След оператора return се задава стойност от указания тип за връщана стойност или тип съвместим с него. Например, ако типът за връщана стойност е double, може върнатата с return стойност да е от тип int; ако типът е клас или интерфейс, трябва да се върне негова инстанция, която може да се създаде и чрез клас-наследник.

```
public static double min(int i1, int i2) {
      return i1<i2 ? i1 : i2;
}
```

При конструкторите не се указва тип на връщана стойност.

```
public class Person {
   // Конструктор
   public Person(String name, int age){
   ...
   }
}
```

Формални параметри на методи

Като формални параметри, при декларацията на методите могат да се описват променливи от всякакви типове, разделени със запетаи. За всеки параметър се задава тип, последван от име. Формалните параметри се ползват като локални променливи в тялото

на метода. Поради това, в тялото не може да се декларират локални променливи с имена като на формалните параметри.

```
public static double max(double d1, double d2) {
      double max = d1>d2 ? d1 : d2;
      return max;
}
```

Задаването на променлив брой формални параметри на метод става с оператор „…”, което е описано в частта Предаване на масиви като параметри и променлив брой аргументи на метод на глава 8. Променлив брой параметри може да се задава само като последен елемент в списъка с формални параметри.

```
public static void varArgs(double d, int ... ints)
```

Извикване на метод с фактически параметри

При извикването на метод, за всеки един от формалните параметри се задава съответен конкретен фактически параметър, като се запазва позиционно съответствие.. Фактическият параметър може да бъде литерал, променлива, функция или израз. Важно е всяка двойка формален-фактически параметър да са от един и същи тип или типовете да са съвместими.

```
public static void main(String[] args) {
  double max1 = max(1, 2);                  // извикване с литерали
  double a = Double.parseDouble(args[0]);
  int b = Integer.parseInt(args[1]);
  double max2 = max(a, b);                  // извикване с променливи
  double max3 = max(max1*2+3, max(a, b)); // извикване с изрази и функции
}
```

При всяко извикване на метод, фактическите параметри се присвояват на съответните формални параметри. Казва се, че фактическите параметри се предават на формалните. Методът се изпълнява с текущите стойност на формалните параметри. След като методът приключи изпълнението си, програмата продължава с изпълнението на следващата команда.

Предаване на параметри по адрес и по стойност

В Java примитивните типове се предават по стойност, а сложните – по адрес.

За примитивните типове, формалният параметър и съответният му фактически параметър заемат самостоятелни клетки от статичната памет. При предаването, стойността на фактическият параметър се записва в паметта за формалния.

За сложните типове като класове и масиви, имената на променливите реферират обекти от динамичната памет. При предаването на параметър, променливата за формален параметър приема стойността на фактическия параметър, която обаче е адрес от динамичната памет. По този начин тя започва да рефери паметта за обекта-фактически параметър.

В Java предаването по адрес на обекти от сложен тип се извършва по подразбиране, но в други езици за програмиране то става чрез използването на допълнителни езикови конструкции.

Промяната на формалните параметри се отразява на фактическите параметри след извикването на метода, ако те се предават по адрес, и не се отразява – ако са предадени

по стойност. Поради тази причина например, в разглежданите методи за сортиране масивите-фактически параметри, предадени по адрес, са пренаредени след изпълнение на методите.

Допълнителни обяснения на описания процес има в частта Предаване (присвояване) на параметри по адрес и по стойност от глава 3.

Страничен ефект

Промяната на стойността на фактически параметър при извикването на метод се нарича страничен ефект от изпълнението на метода.

Предаването по адрес спестява заделянето на излишна памет, но могат да възникнат проблеми, свързани с възможната промяна на формалния и съответно на фактическия параметър.

Страничният ефект не винаги е очакван от потребителите. Стандартната логика е методите, които връщат резултат, да се дефинират като функции с тип на връщана стойност.

Промяната на фактическия параметър може да е нежелана от потребителите. Ако желаят да запазят началната стойност на фактическия параметър, който се изменя, потребителите сами трябва предварително да създадат нова променлива.

При създадените от нас методи за сортиране е ползван страничен ефект за масивите. Тъй като не е указван тип на връщана стойност, страничният ефект не е чак толкова заблуждаващ.

Клониране на обекти

Ако е необходимо да се имитира предаване на сложен обект по стойност, трябва да се създаде нов обект, да се дублират в него стойностите на полетата на оригиналния обект, и след това да се работи с новия обект. Не винаги е необходимо програмистът да реализира логиката за копиране. Може да се клонират автоматично обекти (включително и масиви) с помощта на метод clone, наследен от класа Object. При създаване на клас, за който е необходима подобна логика, трябва да се презапише методът clone. В следващия пример е показано само използването на метода за клониране:

```
public static int[] sort(int [] arr) {
      // int [] newArray = arr; - логическа грешка
      // така newArray и arr са референции към един и същи обект

      // Клониране - създава се нов масив, в него се копира стария.
      // newArray реферира новия
      int [] newArray = arr.clone();

      //... работа с newArray

      return newArray;
}
```

Сигнатура на метод

В езиците за програмиране сигнатура на метод е част от декларацията му, включваща името на метода и някои негови характеристики. Тя е уникално описание, по което методът се отличава от всички останали методи.

В Java сигнатурата включва само името на метода и параметрите му. Може да се дефинират множество методи с уникални сигнатури – с едно и също име, но с различни по брой и/или тип параметри. Сигнатурата не включва модификатори, типа на

връщаната стойност и списъка с изключения. В следващите примерни декларации на методи сигнатурите са оцветени в червено.

```
public static int method1(int i, double d)
public static void method2()

public static void test()
public static void test(int i)
public static void test(long i)
public static int test(int i1, int i2)
```

Декларациите са валидни. Невалидна би била например, декларация „int test()”, защото типът не е част от сигнатурата и метод „test()” вече съществува. При дублиране на сигнатура, компилаторът извежда съобщение за грешка.

Предефиниране (overloading) на методи

Методи от един клас, с едно и също име, но с различни сигнатури, се наричат предефинирани.

При обръщение към метод, за който има няколко дефиниции, в зависим от типа на фактическите параметри автоматично се извиква най-подходящият. Ако не съществува метод с точните съответни типове за формалните параметри, се търси „най-близкият” метод, за който фактическите параметри могат да бъдат преобразувани до типа на формалните.

Тъй като някои параметри не може или не е удачно да се преобразуват по тип, в множество класове са дефинирани методи с едно и също име и различни по тип параметри. Напр., има множество методи System.out.println – за всеки един от примитивните типове, за обект от клас String или Object, за масив от символи, а също и без параметри; има четири метода Math.max – за параметри с типове int, long, float, double и много други.

В следващия пример са предефинирани няколко метода, а в main метода са извикани с различни по тип параметри.

```
// Еднакви методи с различни сигнатури
public class OverloadingTest {
      public static void max(int param1, int param2){
            int max = param1>param2?param1:param2;
            System.out.println("max(int параметри): " + max);
      }
      public static void max(long param1, long param2){
            long max = param1>param2?param1:param2;
            System.out.println("max(long параметри): " + max);
      }
      public static void max(float param1, float param2){
            float max = param1>param2?param1:param2;
            System.out.println("max(float параметри): " + max);
      }
      public static void max(double param1, double param2){
            double max = param1>param2?param1:param2;
            System.out.println("max(double параметри): " + max);
      }

      public static void main(String[] args) {
            max(5, 10);       // int параметри
            max(5, 10L);      // int и long параметри
            max(2.1F, 3.1F); // float параметри
```

```
            max(3.2, 1);      // double и int параметри
      }
}

Резултат:
max(int параметри): 10
max(long параметри): 10
max(float параметри): 3.1
max(double параметри): 3.2
```

Задаването на параметри със стойности по подразбиране е част от механизма за предефиниране на методи в различни езици за програмиране. **В Java не могат да се задават параметри със стойности по подразбиране.**

Модификатор static - статични и динамични методи

Методи, за които е зададен модификатор static, са статични, а такива, за които не е зададен, са динамични.

Динамичните методи могат да се използват единствено чрез инстанция на класа, а статичните – без инстанция.

Когато методите се изпълняват по еднакъв начин, без да се обръщат към специфичните за обектите полета, е подходящо да се декларират като статични. Повечето разгледани досега примери са такива – при методите за работа с числа и масиви не се използват полета на класа, а се задават като параметри. Статичен метод може да се обръща към полета с модификатор static и към други статични методи без да е необходимо да се създава инстанция на класа. Използвали сме например, статичното поле PI, както и статични методи (като sqrt, pow и др.) от класа Math.

Динамични методи са необходими, когато резултатът им зависи от полетата на конкретния обект. Напр. различните обекти от клас Човек имат различни стойности за полетата име и възраст. Съответно, методите се изпълняват върху различни конкретни стойности за полетата.

Други модификатори за методи

Ако се използва модификатор **strictfp,** всички операции с плаваща запетая ще се изпълняват при стриктното спазване на стандарта IEEE 754. Използването на strictfp гарантира, че резултатите от действията с плаваща запетая ще са еднакви при изпълнението на програмата за различните платформи. В противен случай, някои от действията се изпълняват с по-бързи платформено зависими алгоритми, които могат да предизвикат незначителни отклонения в резултата.

Модификаторът **abstract** указва, че методът е абстрактен. Абстрактният метод само се декларира, без да се описва тялото му. Тялото обаче трябва да бъде написано в класовете наследници. Ако даден клас притежава абстрактен метод, класът също се декларира с ключовата дума abstract. Инстанция на абстрактен клас може да бъде създадена единствено чрез наследник, който не е абстрактен и не може да бъде създадена директно чрез конструктор на абстрактния клас.

Модификаторът **final** указва, че методът не може да бъде презаписан в наследник т.е. в наследниците не може да съществува метод със същата сигнатура.

Модификаторите abstract и final са свързани с обектно-ориентираните концепции, които ще бъдат разгледани по-подробно в следващи части на книгата.

Библиотеката Java Native Interface (JNI) позволява изпълнението на код, написан на друг език за програмиране – C, C++, Assembler, Fortran. Методи на Java, включващи такъв код се декларират с модификатор **native**. Методи, написани на други езици се

ползват, за да се ускори изпълнението на критични за програмата дейности. Недостатък е, че по този начин програмата става зависима от платформата, за която е написан външния метод. В Java има и други клиент-сървър механизми за изпълнение на външен код (Sockets, CORBA, Web Serices и др.), при които сървърите или клиентите може да са написани на произволни езици.

Модификаторът **synchronized** се използва за синхронизиране на работата с нишки (клас java.lang.Thread). Когато в обект се чете и записва едновременно от различни нишки, може да възникнат проблеми с консистентността му – състояние, при което данните са валидни. Синхронизирането решава тези проблеми с помощта на различни механизми за конкурентно (Concurrency) управление.

Рекурсия

В тялото на един метод може да има обръщение към различни други методи.

При рекурсията един метод се обръща към себе си. Метод, използващ рекурсия, се нарича рекурсивен.

Пряка рекурсия е, когато **един метод** в тялото си **директно извиква себе си.**

Непряка (косвена) рекурсия има, когато **един метод извиква** себе си индиректно чрез **втори метод, който от своя страна извиква първия**. В този случай и двата метода се извикват рекурсивно. Възможно е в непряката рекурсия да участват повече от два метода.

Рекурсивното извикване на методи не трябва да е безкрайно. Необходимо е да има добре дефинирано **условие за край на рекурсията**, което **зависи от параметрите на метода**. По време на изпълнението на методите, стойностите на параметрите трябва да се променят, така че в определен момент да се достигне условието за край. В този случай се казва, че е достигнато дъното на рекурсията.

С рекурсия могат да бъдат реализирани алгоритми, при които част от решението включва използването на същите алгоритми, но приложени върху „по-елементарни" данни. Лесно, но не винаги ефективно, се намират членовете на числови редици, описани чрез рекурентни формули. Рекурсивните алгоритми са близки до реалната логика и понякога спестяват писането на много код. За да бъдат ефективни, рекурсивните методи не трябва да отнемат много памет и процесорно време.

Рекурентни формули

В различни области на науката се ползват редици от числа, за които са зададени стойностите на първите няколко елемента, а всеки следващ се получава чрез предходните. Редици, описвани с рекурентни формули, са:

- аритметична прогресия – първият елемент е число a_1, а следващите се изчисляват по формулата $a_n = a_{n-1} + b$, за $n>1$, където b е разлика на прогресията;
- факториел – първи елементи са 0!=1 и 1!=1, следващите се изчисляват по следния начин: $n! = n*(n-1)!$, за $n>1$;
- редица на Фибоначи – първите два елемента са 0 и 1, следващите се изчисляват по формулата: $a_n= a_{n-1}+ a_{n-2}$, за $n>1$;
- и мн. др.

За край на рекурсията се използват условия, определени върху първоначално зададените елементи, които не се описват с рекурентната формула.

За да се намери стойността на n-тия елемент от редицата, се прилага многократно същата формула, но за по-малки стойности на елементите – (n-1), (n-2),…,1,0. За да се получи търсеният резултат, всички предходни стойности трябва да се изчислят и акумулират в него.

Реализация на рекурсия

В решените задачи към глава 8 (намиращи се в архивния файл с примери към книгата) са разгледани методи за намиране на факториел и елемент от редицата на Фибоначи.

Рекурсивна реализация на факториел е дадена в следния примерен код:

```
public static long f(int n){
      if(n<2) return 1;

      return n*f(n-1);
}
```

Методът получава като параметър число n, чийто факториел трябва да се изчисли. В резултат връща n умножено с f(n-1). При извикването на f(n-1), той от своя страна ще извика f(n-2) и т.н. докато параметъра при някое от извикванията стане равен на 1. Тогава условието „n<2” ще стане истина и ще се изпълни тялото на условния оператор if, като по този начин ще приключи изпълнението на рекурсията.

Особеност при рекурсията е, че първоначално извикания метод f(n) не приключва, докато не приключи извикания от него f(n-1), той от своя страна не приключва докато не приключи f(n-2) и т.н. до f(1), който приключва с върнат резултат 1. Така, в определен момент ще имаме n метода за факториел, които са стартирани едновременно.

При изпълнението си, всеки метод работи с част от стека на паметта, в която са описани работните му променливи и операции върху тях (оператори, конструкции, методи и т.н.). Ако има едновременно стартирани много методи, стека, контролиран от ВМ на Java може да се препълни поради недостиг на памет. В такъв случай се получава грешка по време на изпълнение на програмата java.lang.StackOverflowError. Това „лесно” може да се постигне с помощта на рекурсивни методи, които се извикват многократно.

Въпреки, че рекурсивните методите са „елегантни”, (с малко код и относително лесно разбираеми), трябва да се внимава с тях, тъй като може да заемат много памет и да се изпълняват бавно. Забавянето идва от необходимостта всеки метод да записва информация в стека при извикването му и да се изчиства от стека при приключване на изпълнението му.

Методът за факториел не е „най-страшният” рекурсивен метод. Нека разгледаме рекурсивната реализация на алгоритъма за намиране на числата на Фибоначи:

```
public static long fib(int n){
      if(n<1) return 0;
      if(n<=2) return 1;

      return fib(n-1) + fib(n-2);
}
```

За да се намери n-тият елемент от редицата, първо трябва да се намерят предходните два. При всеки от тях методът се извиква отново по два пъти и т.н., докато не се стигне края на рекурсията, зададен чрез два условни оператора. За разлика от рекурсивния метод за факториел, при който има линейна рекурсия, тук рекурсията е разклонена (дървовидна) и при всяка следваща стъпка броят на извикваните методи се удвоява (за тези, които не са стигнали края на рекурсията): fib(n)→ fib(n-1), fib(n-2) → fib(n-2), fib(n-3), fib(n-3), fib(n-4)... При втората стъпка, за да се изчисли fib(n-1), трябва да се изчислят fib(n-2) и fib(n-3), а за fib(n-2) – fib(n-3) и fib(n-4). Вижда се, че въпреки дублираните извиквания на методи с едни и същи параметри, при всяко изпълнение те се изчисляват самостоятелно. За да се намери 10-тият член на редицата, методът ще се изпълни 109

пъти, 11-тия – 177, 12-тия – 287,...30-тия – 1 664 079, а 40-тия 204 668 309 пъти и дори ще се наложи да изчакаме известно време. Това преброяване може да се направи чрез добавяне на статично поле за брояч, който се увеличава с единица при всяко извикване на метода:

```
public class Fibonachi {
  static long count; // брояч за изпълненията на рекурсивния метод
  public static long fibonachiRecursion(int n){
    count++;
    if(n<1) return 0;
    if(n<=2) return 1;

    return fibonachiRecursion(n-1) + fibonachiRecursion(n-2);
  }

  public static void main(String[] args) {
    int n = 40;
    long fibR = fibonachiRecursion(n);

    System.out.println(n + "-ият член на редицата на Фибоначи е "+ fibR +
        ". Брой изпълнения на метода: "+ count);
  }
}

Резултат:
40-ият член на редицата на Фибоначи е 102334155. Брой изпълнения на метода:
204668309
```

Кое да изберем: рекурсия, итерация, формула, съхраняване на резултатите?

Итеративните решения са малко по-трудни за програмиране. При тях трябва да се помисли върху алгоритъма, да се добавят допълнителни локални променливи и да се използва цикъл. Когато паметта и бързодействието са важни, итеративните решения са по-добри от рекурсивните, а що се отнася до редицата на Фибоначи, очевидно използването на рекурсивния метод е доста неефективно.

Итеративен метод за намирането на факториел, заедно с вече показания рекурсивен метод, е реализиран в следващия клас:

```
public class Factorial {
      // рекурсивен метод
      public static long factorialRecursion(int n){
            if(n<2) return 1;

            return n*factorialRecursion(n-1);
      }

      // итеративен метод
      public static long factorialIteration(int n){
            long factorial = 1;

            for(int i=2; i<=n; i++){
                  factorial = i*factorial;
            }
            return factorial;
      }

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

```
            int n = 5;
            long factR = factorialRecursion(n);
            System.out.println(n + "!= "+ factR);
      }
}
```

Сложността на реализация на итеративния метод не е чак толкова голяма: в един цикъл се променя управляващата променлива от 2 до n със стъпка 1 и при всяка итерация се умножава с текущата стойност на променливата за факториел.

При итерационното решение за редицата на Фибоначи се използват 3 променливи - за n-тия и за предходните два елемента. В цикъл се изчислява n-тия елемент и се подготвят другите два елемента за следващата итерация. Както и в рекурсивния вариант, цикълът не се изпълнява за началните условия.

```
public static long fibonachiIteartion(int n){
      int an=0;     // елемент n
      int an_1=1;   // елемент n-1
      int an_2=1;   // елемент n-2

      if(n<1) an=0;
      else if(n<=2) an=1;

      // изчисляване на новата стойност на an
      // и следващите стойности на an_2 и an_1
      for(int i=3; i<=n; i++){
            an = an_1+an_2;
            an_2 = an_1;
            an_1 = an;
      }

      return an;
}
```

Реализираният итеративен вариант е с линейна сложност, докато рекурсивният е с експоненциална.

Друг подход при търсене на членове на редици е да се намери аналитична формула за n-тия елемент. В някои от случаите това е невъзможна или трудна задача, в други - формулата е сложна за изчисление. Например, факториел няма аналитичен вариант.

Аналитичната формулата за намиране на елемент от редицата на Фибоначи е следната:

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Формулата очевидно, дава като резултат реално число, което се нуждае от закръгляване при програмната реализация:

```
      public static long fibonachiFormula(int n){
            return Math.round( ( (Math.pow((1+Math.sqrt(5))/2, n)
                                    + Math.pow((1-Math.sqrt(5))/2, n)
                                  )
                                  /Math.sqrt(5)
                                )
                        );
      }
```

Тази формула е удачно да се ползва, когато се изчислява един елемент от редицата на Фибоначи. Операциите по коренуване, повдигане на степен, деление и закръгляване обаче, отнемат доста процесорно време. Трябва да се прецени кой от всички варианти да се използва, когато се търси множество от различни елементи.

По-елементарен пример за рекурентен и аналитичен вариант на формула е за намиране n-тия член на аритметична прогресия. Рекурентната формула за аритметична прогресия е $a_n = a_{n-1} + b$. Алгоритъмът, прилагащ тази формула, е с линейна сложност. Ако обаче се използва аналитичната формулата за общия член $a_n = a_1 + (n-1)*b$, сложността на алгоритъма ще е константна и изчислението няма да отнема много процесорно време.

В решените задачи към глава 8 (може да свалите архивния файл от адрес http://fmi.fractime.net/javabook/javabook.rar), са показани и други итеративни решения за намиране на факториел и редицата на Фибоначи, при които елементите на редиците са записани в масив. Всеки следващ елемент използва вече изчислените и записани в масива предходни елементи. По този начин, в една програма може многократно да се търсят различни елементи от редиците, както и да се извършват допълнителни обработки върху тях, без да е необходимо да се изчисляват всеки път. В следващия код е даден варианта със запазване на стойностите в масив – метод fibonachiArray – и класът с всички предложени решения:

```
// Различни решения за намиране на n-тия елемент
// от редицата на Фибоначи
public class Fibonachi {
	static long count; // брояч за изпълненията на рекурсивния метод

	// за запис на елементите в масив при метода fibonachiArray
	static final int maxFibonachiLength = 90;
	static long [] fibArr = new long[maxFibonachiLength];

	// рекурсивно решение
	public static long fibonachiRecursion(int n){
		count++;
		if(n<1) return 0;
		if(n<=2) return 1;

		return fibonachiRecursion(n-1) + fibonachiRecursion(n-2);
	}

	// итеративно решение
	public static long fibonachiIteration(int n){
		int an=0;    // елемент n
		int an_1=1;  // елемент n-1
		int an_2=1;  // елемент n-2

		if(n<1) an=0;
		else if(n<=2) an=1;

		// изчисляване на новата стойност на an
		// и следващите стойности на an_2 и an_1
		for(int i=3; i<=n; i++){
			an = an_1+an_2;
			an_2 = an_1;
			an_1 = an;
		}

		return an;
```

```
	}

	// решение с аналитична формула
	public static long fibonachiFormula(int n){
		return Math.round( ( (Math.pow((1+Math.sqrt(5))/2, n)
					+ Math.pow((1-Math.sqrt(5))/2, n)
				  )
				  /Math.sqrt(5)
				)
			);
	}

	// ползване на масив за съхранение на елементите
	public static long fibonachiArray(int n){

		if(n==0) return 0;
		if(fibArr[n]>0)          // ако вече сме попълвали масива
			return fibArr[n]; // връщаме елемента

		// иначе попълваме масива еднократно
		fibArr[0] = 0;
		fibArr[1] = 1;

		for (int i = 2; i < fibArr.length; i++) {
			fibArr[i]=fibArr[i-2]+fibArr[i-1];
		}

		return fibArr[n];
	}

	public static void main(String[] args) {
		int n = 10;

		System.out.println(n + "-ият член на редицата на Фибоначи е:
");

		count = 0;
		long fibR = fibonachiRecursion(n);
		System.out.println("рекурсия: "+ fibR
				+ ". Брой изпълнения на метода: "+ count);

		long fibF = fibonachiFormula(n);
		System.out.println("формула: "+ fibF);

		long fibA = fibonachiArray(n);
		System.out.println("масив: "+ fibA);

		long fibI = fibonachiIteration(n);
		System.out.println("итерации: "+ fibI);
	}
}
```

Резултат:

```
10-ият член на редицата на Фибоначи е:
рекурсия: 55. Брой изпълнения на метода: 109
формула: 55
масив: 55
итерации: 55
```

В случая не трябва да се забравя предложеният вариант към решенията на задачите в глава 8, при който се използва BigInteger вместо long за елементите на масива. В променлива от тип long може да се запише само до около стотния елемент на редицата на Фибоначи.

Приложения на рекурсията

Рекурсията може да се използва за създаване на елегантни решения на различни проблеми. Разбира се, от тях изключваме показаните рекурсивни методи за намиране на факториел и числа на Фибоначи. Някои от полезните приложения са за стандартни елементарни операции, а други в различни научни области:

- работа с дървета, включваща добавяне, премахване, промяна на възли и търсене – при тях възелът е обект, съхраняващ информация (в полета на клас) за свързаните с него други възли;
- търсене на път, което е частен случай на търсенето в дървета;
- бързо сортиране QuickSort;
- сортиране чрез сливане MergeSort;
- двоично търсене;
- фрактали;
- генетични алгоритми;
- невронни мрежи и др.

Рекурсивен алгоритъм за двоично търсене в подреден масив

Рекурсивната реализацията на двоично търсене в сортиран масив не се различава като ефективност от итеративния вариант, показан в глава 8, Двоично търсене. Като параметри на метода се задават масива, търсената стойност и индексите на началния и крайния елементи на масива, измежду които се извършва търсенето. В тялото на метода, ако средният елемент е търсеният, методът връща неговия индекс. В противен случай рекурсивно се извиква търсенето за лявата или дясната част, от елементите на масива. Ако не се намери резултат (когато левия индекс стане по-малък от десния), методът връща стойност -1. Последно извиканият метод ще върне -1 или намерената стойност към предходно извикалия го метод, той от своя страна ще върне стойността на по-горния метод и т.н., до първоначално извикания метод. Реализацията на тази логика е показана в следващия клас:

```
public class BinarySearchRecursive {

  public static int binarySearchRecursive(int[] arr, int left,
                                          int right, int searchVal) {
    if (left < right) {
       int middle = left + (right - left) / 2;

       if (searchVal == arr[middle]) {
         return middle;
       }

       if (searchVal < arr[middle]) {
         // продължаваме търсенето от елемент left до middle
         return binarySearchRecursive(arr, left, middle, searchVal);
       } else {
         // продължаваме търсенето от елемент middle + 1 до right
         return binarySearchRecursive(arr, middle + 1, right, searchVal);
       }
    }
```

```
    return -1;
  }

  public static void main(String[] args) {
    // Сортиран масив
    int[] arr = { 1, 5, 8, 16, 30 };

    int searchVal = 16;
    // рекурсията започва с търсене в масива от елемент 0 до arr.length
    int searchPos = binarySearchRecursive(arr, 0, arr.length, searchVal);
    System.out.println("Елемент " + searchVal + (searchPos == -1 ? " не е намерен"
: " е намерен на позиция " + searchPos));

    searchVal = 2;
    searchPos = binarySearchRecursive(arr, 0, arr.length, searchVal);
    System.out.println("Елемент " + searchVal + (searchPos == -1 ? " не е намерен"
: " е намерен на позиция " + searchPos));
  }
}
```

Резултат:

```
Елемент 16 е намерен на позиция 3
Елемент 2 не е намерен
```

Въпроси и задачи за упражнения

1. Тествайте модификаторите за достъп.
2. Изпробвайте модификаторите abstract, final, native, synchronized, strictfp.
3. Създайте рекурсивно решение с линейна сложност за намиране на числата от редицата на Фибоначи.
4. Реализирайте рекурсивен метод за бързо сортиране на масив от цели числа.
5. Редици от числа се описват по следния начин:
   а. $a_n = 1/n$, за $n>=1$
   б. $a_n = a_{n-1} + n$, $a_0=0$, $n>=0$
   в. $a_n = 2*a_{n-1} - a_{n-2} + a_{n-3}$, $a_0=0$, $a_1=1$, $a_2=2$, $n>=0$

   Създайте различни методи (итеративни, рекурсивни и др.) за намиране на i-тия елемент и сумата на първите m елемента на редицата.

Упътвания към задачите

1. Създайте два класа в различни пакети. В тях задайте полета и/или методи с различни модификатори за видимост. В единият от пакетите направете трети клас-програма, в който създавате обекти от класовете и се опитвате да достъпите полетата и методите на първите два класа.
2. Някои от модификаторите ще бъдат разгледани подробно в следващата част на книгата. Потърсете примери с модификаторите в Интернет и на тяхна база реализирайте свои собствени.
3. За намиране на n-тия елемент използвайте масив с n елемента, който предавате като параметър на рекурсивния метод.
4. Потърсете подобна реализация в Интернет и я проучете.
5. Използвайте методите за числата на Фибоначи и ги променете.

Глава 11. Символни низове

Символните низове са последователности от символи, включващи букви на различни езици, цифри, пунктуационни знаци и други. В много практически задачи се налага обработката на низове. За някои по-елементарни операции върху низове са създадени стандартни методи като: сравнения на низове; преобразуване на низ в цяло или реално число; търсене по ключови думи в низ; търсене по шаблони, в които части от търсените низове са заместени със символи маски; заместване на един низ с друг низ. Чрез използването на стандартните методи могат да се решават много по-сложни задачи, напр. за автоматични преводи между различни естествени езици, автоматизирано генериране на разнотипни структурирани документи, извличане на мета данни от структурирани или не структурирани, но от определена предметна област документи. Структурираните документи могат да бъдат от различни стандартни формати – XML, CSV, SQL, JSON, YAML и др. – или не толкова добре структурирани като HTML, PDF и др. Процесът на компилиране също е свързан с разпознаване на записаните в изходния текст на програмата езикови конструкции и преобразуването им до машинни инструкции в изпълнимия файл (който също е вид текст).

В тази глава са представени стандартни класове и методи за работа с низове в Java. Дадени са примери за употребата им при решаването на различни задачи.

Символни низове в Java

Всички низове в Java са представители на класа java.lang.String. Низовете са последователности от UNICODE символи от тип char. Обектите от класа String са непроменяеми (immutable) т.е. действията върху обект не променят самия обект за низ, а създават нов низ.

Низ, зададен като литерал – напр. `"текст1"` – притежава съответен обект от класа String в динамичната памет. Обикновено низовете се асоциират с променливи, чрез които може да се обработват допълнително. В следващия пример, променливата s реферира създаден в динамичната памет низ, съдържащ стойността `"текст2"`.

```
String s = "текст2";
```

Тъй като низовете са често използвани, за улеснение на програмистите е реализирано едно изключение в езика: може да се създават обекти – низове чрез литерал, без явно използване на конструктор.

Създаването на обект от класа String може да се извърши и по стандартния за всички обекти начин – чрез конструктор на класа. Има няколко конструктора с различни параметри: низ, който може да е променлива или литерал, масив от символи (от тип char), обекти от клас StringBuffer или StringBuilder и др. Обекти от клас String може да се създадат например по следния начин:

```
String s1 = new String("низ1");        // параметър низ-литерал
char [] chars = {'н', 'и', 'з', '2'}; // масив от символи
String s2 = new String(chars);         // параметър масив от символи
String s3 = new String(s2);            // параметър низ-литерал
```

В класа String, низът се съхранява в поле, което е масив от символи. Това може да се види в сорс кода на класа String. Изходните кодове на Java класовете са в архивния файл src.zip, намиращ се в инсталационната директория на JDK.

Между обектите от класа String, създадени чрез конструктор и литерал има разлика:

- множество от низове, зададени чрез еднакви литерали, използват обща памет;
- всеки низ, създаден с конструктор, заема собствена динамична памет.

Нека например са дадени следните дефиниции на променливи за низове:

```
String s1 = "текст";
String s2 = "текст";
String s3 = new String("текст");
String s4 = new String("текст");
```

Съответните на четирите променливи низове са равни. Обаче, s1 и s2 реферират един и същи адрес, а s3 и s4 – собствени. Компилираният байт код е оптимизиран, така че еднаквите литерали имат единствено представяне.

Сравнение на референции към низ

Твърденията от последния пример може да бъдат проверени чрез **операторите == и != за сравнение на референции.**

```
public static void main(String[] args) {
      String s1 = "текст";
      String s2 = "текст";
      String s3 = new String("текст");
      String s4 = new String("текст");

      System.out.println("s1==s2 -> " + (s1==s2));
      System.out.println("s1==s3 -> " + (s1==s3));
      System.out.println("s3==s4 -> " + (s3==s4));

      System.out.println("s1!=s2 -> " + (s1!=s2));
      System.out.println("s1!=s3 -> " + (s1!=s3));
      System.out.println("s3!=s4 -> " + (s3!=s4));
}

Резултат:
s1==s2 -> true
s1==s3 -> false
s3==s4 -> false
s1!=s2 -> false
s1!=s3 -> true
s3!=s4 -> true
```

Използването на оператори за сравнение при низове дава информация само за това дали референциите сочат към един и същи адрес.

От примера се вижда, че дори два низа да имат едно и също съдържание, сравнението с оператора „==” не ги определя като еквивалентни.

За проверка на това, дали два низа са равни, се използва методът equals на класа String.

Сравнение за равенство на низове с метод equals

Чрез метода equals(низ) на класа String може да се сравни дали текущия низ е равен на зададения като параметър низ. На вътрешно ниво, методът сравнява символ по символ елементите на двата масива, представящи низовете. В следващия пример, към примерните еднакви низове s1 до s4, се добавя друга променлива – s5, съдържаща като низ друг текст. Върху тях се прилага методът equals, който връща като резултат от

сравнението булева стойност. Обектите реферирани от s1 до s4 са равни помежду си, и не са равни на s5:

```
public static void main(String[] args) {
      String s1 = "текст";
      String s2 = "текст";
      String s3 = new String("текст");
      String s4 = new String("текст");
      String s5 = new String("друг текст");

      System.out.println("s1.equals(s2) -> " + s1.equals(s2));
      System.out.println("s1.equals(s3) -> " + s1.equals(s3));
      System.out.println("s3.equals(s4) -> " + s3.equals(s4));
      System.out.println("s1.equals(s5) -> " + s1.equals(s5));
}

Резултат:
s1.equals(s2) -> true
s1.equals(s3) -> true
s3.equals(s4) -> true
s1.equals(s5) -> false
```

Сравнение на символни низове

Методът equals на класа String сравнява дали два низа са равни, като отчита регистъра на буквите, т.е. методът прави разлика между малки и главни букви. За методи, които правят разлика между малки и главни букви се казва, че са case sensitive, а методи, които не правят разлика – са case insensitive.Например, при сравняване на низовете “текст” и “Текст”, методът equals() ще върне резултат false.

Има **няколко метода за сравнение на низове**:

- ***boolean equals(низ)*** – сравнява дали текущият низ е равен на зададения като параметър низ, като се отчита регистърът на буквите. Връща true, ако низовете са равни, иначе – false.
- ***boolean equalsIgnoreCase(низ)*** – логиката му е като на equals, но регистърът на буквите е без значение.
- ***int compareTo(низ)*** – сравнява лексикографски – буква по буква – текущия и низът, зададен като параметър. Връща число: 0 – ако низовете са равни; иначе –разликата между кодове на първите различни символи. Разликата е отрицателно число, ако текущият низ е лексикографски пред низа-параметър и положително – ако текущият низ е лексикографски след параметъра. Регистърът на буквите се отчита.
- ***int compareToIgnoreCase(друг_низ)*** — логиката му е като на compareTo, но регистърът на буквите не е от значение.

В следващия пример е създаден метод compareMethods, получаващ два низа като параметър и демонстриращ методите за сравнение. Използван е в основният метод с различни конкретни параметри.

```
public class StringComparision {
  // Прилагане на методи за сравнение върху низовете s1 и s2
  public static void compareMethods(String s1, String s2) {
    System.out.println('"' + s1 + '"' + ".equals("
            + '"' + s2 + '"' + ") -> " + s1.equals(s2));
    System.out.println('"' + s1 + '"' + ".equalsIgnoreCase("
        + '"' + s2 + '"' + ") -> " + s1.equalsIgnoreCase(s2));
    System.out.println('"' + s1 + '"' + ".compare("
        + '"' + s2 + '"' + ") -> " + s1.compareTo(s2));
```

```
    System.out.println('"' + s2 + '"' + ".compare("
        + '"' + s1 + '"' + ") -> " + s2.compareTo(s1));
    System.out.println('"' + s1 + '"' + ".compareIgnoreCase("
      + '"' + s2 + '"' + ") -> " + s1.compareToIgnoreCase(s2));
  }

  public static void main(String[] args) {
    compareMethods("текст", "Текст");
    compareMethods("аб", "ав");
  }
}

Резултат:
"текст".equals("Текст") -> false
"текст".equalsIgnoreCase("Текст") -> true
"текст".compare("Текст") -> 32
"Текст".compare("текст") -> -32
"текст".compareIgnoreCase("Текст") -> 0
"аб".equals("ав") -> false
"аб".equalsIgnoreCase("ав") -> false
"аб".compare("ав") -> -1
"ав".compare("аб") -> 1
"аб".compareIgnoreCase("ав") -> -1
```

Обхождане елементите на низ

Низът се съхранява в обект на класа String като масив от символи. Тъй като полето е декларирано като private, отделните символи не може да се достъпват директно (чрез правоъгълни скоби и индекс), а се достъпват чрез специален метод – charAt(позиция). Броят на елементите на низ се взима с метод length(), като първият символ от низа, както и при масивите, е с индекс 0. Индексът на последния елемент е length()-1.

В следващия код е създаден метод, с логика подобна на equals(), в който е показано обхождането на елементите на низ. Той е статичен и изисква два низа като параметри. Сравнява съответните символите един по един и връща като резултат булева стойност. Тук е реализирана и друга логика, която може да спести обхождането: ако параметрите низове реферират един и същ обект, значи те са равни, а ако имат различна дължина – са различни.

```
public class ScanString {
  // Сравнява дали низовете s1 и s2 са равни,
  // без използване на стандартен метод equals.
  public static boolean equals(String s1, String s2){
    if(s1==s2){
      return true; // ако променливите реферират един и същ обект
    }

    if(s1.length()!=s2.length()){
      return false;  // ако дължините на низовете са различни
    }

    int count = s1.length();
    for(int i=0; i<count; i++){
      if(s1.charAt(i)!=s2.charAt(i)){
        return false; // ако i-тите елементи са различни
      }
    }

    return true; // ако всички елементи са равни
```

```
  }

  public static void main(String[] args) {
    System.out.println(equals("низ1", "низ1"));  // равни референции
    System.out.println(equals("низ1", "низ12")); // различни дължини
    String s = new String("низ1");
    System.out.println(equals("низ1", s));       // равни елементи
  }
}

Резултат:
true
false
true
```

Конкатениране на символни низове

Низове се конкатенират (съединяват) с помощта на два оператора: + и +=. Те са използвани многократно в различни примери в книгата. Чрез оператора +, към левия операнд се долепва десният. За да бъде крайният резултат низ, поне единият операнд трябва да е низ. При операторът +=, към стоящия отляво низ се долепя десният. В този случай обаче, левият операнд задължително е променлива от тип String и крайният резултат се записва в нея. Използването на оператора + не изисква крайния резултат да бъде записан в променлива. Операторите + и += са улеснения на езика за работа с низове. Логиката на методът concat() на класът String е подобна – прилага се върху текущия низ, към който се долепва низът, зададен като параметър. Той позволява обаче работа както с променливи, така и само с низови литерали (които са обекти от класа String). Например:

```
System.out.println("12" + "34");       // +: конкатенация без присвояване
String s1 = "аб" + "вг";               // +: конкатенация и присвояване
System.out.println(s1);
s1 += "де";                            // +=: конкатенация с присвояване
System.out.println(s1);
System.out.println("56".concat("78"));// +=: конкатенация с/без присвояване

Резултат:
1234
абвг
абвгде
5678
```

Както споменахме, низовете са непроменяеми (immutable). Това означава, че за низовете, получени в резултат от конкатенация, се заделя нова памет. Това е демонстрирано в следващия пример, като за целта са създадени две променливи, реферирящи един и същи низ. При промяна на едната референция чрез конкатенация, другата остава непроменена. В резултат променливите реферират различни адреси.

```
String s2 = "аб";
String s3 = s2;
System.out.println(s2==s3); //true: s2 и s3 реферират един и същи обект
s2 += "вг";
System.out.println(s2==s3); //false: s2 и s3 вече реферират различни обекти
System.out.println(s2);     // абвг
System.out.println(s3);     // аб

Резултат:
true
```

```
false
абвг
аб
```

Ако към даден обект не сочи променлива, то паметта, използвана от него се освобождава автоматично с помощта на механизма Garbage Collector. Стартирането на Garbage Collector обаче, е периодично и се задейства от алгоритми на ВМ на Java. Когато се налага извършването на огромен брой конкатенации (например в цикъл), не е добре да се използва класът String, защото може да се получи препълване на паметта, преди да успее да се стартира Garbage Collector. Вместо String, в такива случаи може да се използват класовете StringBuffer и StringBuilder. При определени операции с литерали, компилаторът на Java прилага оптимизации, използващи StringBuilder вместо String.

Конкатенация със StringBuffer и StringBuilder

Класовете StringBuffer и StringBuilder са променяеми (mutable). StringBuilder, за разлика от StringBuffer, не гарантира синхронизация при използване в множество нишки. Подобно на класа String и двата класа съхраняват низа в масив от символи, който обаче не е деклариран като final и при необходимост увеличава капацитета си.

Конструкторите на StringBuffer и StringBuilder приемат като параметър низ – обект от класа String. Има и друг конструктори – например такъв, в който се указва първоначалният капацитет на буфера. Чрез методи append (променлива от произволен тип), съответния на параметъра низ се добавя след последния символ на буфера. За да се получи от обект от класовете StringBuffer и StringBuilder съдържащият се в него низ, се използва методът toString(). В следващия примерен код е показан начин за работа със StringBuffer. При StringBuilder може да се използват аналогични методи.

```
public class StringBufferAppend {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("аб-"); // нов обект
            sb.append(true);    // добавяме булева стойност
            sb.append("-");     // добавяме низ
            sb.append(12);      // добавяме цяло число

            String result = sb.toString(); // взимаме съдържащия се низ
            System.out.println(result);
      }
}

Резултат:
аб-true-12
```

Методи за преобразуване в малки и главни букви

Методът toUpperCase() връща текущия низ, преобразуван в низ, състоящ се само от главни букви. Методът toLowerCase() извършва преобразуване в низ, съдържащ само малки букви. Например:

```
String s = "аБВг";
System.out.println(s.toLowerCase());
System.out.println(s.toUpperCase());
System.out.println(s);
// Припомняме, че низът не се променя при действията върху него,
// а методите връщат нов низ.

Резултат:
```

```
абвг
АБВГ
аБВг
```

Методи за търсене в низ

Методите за търсене в низ дават информация за това, дали зададен като параметър низ се среща в текущия низ. Някои от методите връщат булева стойност (среща се / не се среща), а други връщат число – позицията, от която започва търсеният подниз или -1 – ако не се среща.

Методи за търсене, връщащи булева стойност:

- ***boolean startsWith(низ_за_търсене)*** – сравнява дали текущият низ започва с низа, зададен като параметър. Връща true, ако търсенето е успешно, иначе – false.
- ***boolean endsWith(низ_за_търсене)*** – сравнява дали текущият низ завършва с низа, зададен като параметър. Връща true, ако търсенето е успешно, иначе – false.
- ***boolean contains(низ_за_търсене)*** – сравнява дали текущият низ съдържа низа зададен като параметър. Връща true, ако търсенето е успешно, иначе – false.

Методите са case sensitive и затова чрез тях може да се намери само точното срещане на подниза. Ако е необходимо да се извърши търсене, без значение от регистъра на буквите, може първо низът да се преобразува до низ с малки или главни букви. В следващия пример е демонстрирана възможна работа с разгледаните методи.

```
public class StringContains {

  // Прилагане на методи за търсене на низ s2 в низ s1
  public static void containsMethods(String s1, String s2) {
    System.out.println('"' + s1 + '"' + ".startsWith("
            + '"' + s2 + '"' + ") -> " + s1.startsWith(s2));
    System.out.println('"' + s1 + '"' + ".endsWith("
        + '"' + s2 + '"' + ") -> " + s1.endsWith(s2));
    System.out.println('"' + s1 + '"' + ".contains("
        + '"' + s2 + '"' + ") -> " + s1.contains(s2));
  }

  // Метод за търсене без значение от регистъра на буквите
  public static void containsIgnoreCase(String s1, String s2) {
    System.out.println('"' + s1 + '"' + ".toLowerCase().contains("
            + '"' + s2 + '"' + ".toLowerCase()) -> "
            + s1.toLowerCase().contains(s2.toLowerCase()));
  }

  public static void main(String[] args) {
    containsMethods("По-дълъг текст", "текст");
    containsMethods("По-дълъг текст", "Текст");
    containsIgnoreCase("По-дълъг текст", "Текст");
  }
}

Резултат:
"По-дълъг текст".startsWith("текст") -> false
"По-дълъг текст".endsWith("текст") -> true
"По-дълъг текст".contains("текст") -> true
"По-дълъг текст".startsWith("Текст") -> false
"По-дълъг текст".endsWith("Текст") -> false
"По-дълъг текст".contains("Текст") -> false
"По-дълъг текст".toLowerCase().contains("Текст".toLowerCase()) -> true
```

Върху методите, които връщат низ, може да се прилагат директно други методи, подобно на дадения пример в метода containsIgnoreCase:

```
s1.toLowerCase().contains("...")
```

Изпълнението на такива „вериги от методи" е отляво-надясно. В случая, първо се изпълнява преобразуването на обекта s1 до низ с малки букви – s1.toLowerCase(). След това, върху получения резултат се изпълнява методът contains. „Верига от методи" може да се създава за всякакви методи, връщащи обекти, върху които може да се прилагат други, валидни за съответния обект методи.

Методи, които при търсене връщат индекс, са:

- ***int indexOf(параметър)*** – параметърът е низ или символ от тип char, който търсим. Методът връща индекс от текущия низ, от който започва първото срещане на указания като параметър търсен низ, или „-1" – ако търсенето е неуспешно.
- ***int indexOf(параметър, позиция)*** – методът е подобен на горния, но търсенето започва от зададената като втори параметър позиция.
- ***int lastIndexOf(параметър)*** – търсенето се извършва от крайния символ към началния.
- ***int lastIndexOf(параметър, позиция)*** – търсенето се извършва от указаната позиция към началото.

В следващия пример се намират всички позиции, на които даден подниз се среща в низ:

```
public class SearchInString {
  // Всички позиции, на които се среща s2 в s1.
  public static void indexOfAll(String s1, String s2) {
    System.out.print("Позиции, на които се среща \""
        + s2 + "\" в \"" + s1 + "\": ");

    int index = s1.indexOf(s2);  // позиция на първото срещане

    while (index >= 0) {         // докато индекса е >=0
      System.out.print(index);   // отпечатване на индекса
      index = s1.indexOf(s2, index + 1); // търсене на следващ
      if(index>=0)               // отпечатване на ", " ако има следващ
        System.out.print(", ");
    }

    System.out.println();
  }

  public static void main(String[] args) {
    indexOfAll("Текст 1. Текст 2. Текст 3.", "Текст");
    indexOfAll("Текст 1. Текст 2. Текст 3.", "текст");
  }
}

Резултат:
Позиции, на които се среща "Текст" в "Текст 1. Текст 2. Текст 3.": 0, 9, 18
Позиции, на които се среща "текст" в "Текст 1. Текст 2. Текст 3.":
```

От примера се вижда, че търсенето при методите indexOf() зависи от регистъра на буквите. Ако е необходимо търсенето да не зависи от регистъра, може да се извърши преобразуване до малки или големи букви на двата низа.

Извличане на подниз

Методът **substring(начален_индекс)** връща подниз на текущия низ, който започва от зададения като параметър начален индекс и съдържа всички символи на текущия низ до края. Методът **substring(начален_индекс, краен_индекс)** връща подниз, включващ символите от зададения като параметър начален_индекс до „краен_индекс-1” т.е. символът за указаният като параметър краен индекс не се включва. Например:

```
String s = "абвгде";
// индекси  012345

System.out.println(s.substring(2));
System.out.println(s.substring(0, 2));
System.out.println(s.substring(2, 4));

Резултат:
вгде
аб
вг
```

Методите за търсене и извличане на поднизове (indexOf и substring) може да се използват за автоматично извличане на информация от структурирани и полу-структурирани документи. В следващия пример се използва входен XML низ, в който има описания на продукти. Низът се парсира и се извеждат само имената на продуктите, като е известно, че имената са описани в таг <name>.

```
public class SimpleTextParse {
  // Извежда имената на продукти, описани в структуриран текст
  public static void main(String[] args) {
    String xml = "<products>"
        + "<product><name>Продукт 1</name><price>Цена 1</price></product>"
        + "<product><name>Продукт 2</name><price>Цена 2</price></product>"
        + "</products>";

    String startTag = "<name>";
    String endTag = "</name>";
    int indexFrom = xml.indexOf(startTag); // Начален индекс за тага <name>
    indexFrom += startTag.length();        // Начален индекс за името
    int indexTo = xml.indexOf(endTag);     // Краен индекс за името

    while (indexFrom >= 0 && indexTo >= 0){
      System.out.println(xml.substring(indexFrom, indexTo));
      indexFrom = xml.indexOf(startTag, indexFrom+1);
      indexFrom += startTag.length();
      indexTo = xml.indexOf(endTag, indexTo+1);
    }
  }
}

Резултат:
Продукт 1
Продукт 2
```

За работа с XML документи в Java са разработени специализирани класове. В примера само е демонстриран начин за елементарно извличане на данни от структуриран текст. При търсенето е известно, че името на продукта се намира между началния таг „<name>” и крайния – „</name>”. Извършва се търсене на техните индекси и към индекса

на началния таг се добавя дължината на низа за този таг. След това се използва методът substring за извличане на подниза, намиращ се между двата индекса. **Трябва да отбележим, че примерът е опростен**, защото обикновено в таговете има атрибути от вида „<таг атрибут$_1$=”стойност$_1$” атрибут$_2$=”стойност$_2$”...>”. Тогава за търсене може да използваме регулярни изрази или малко по-сложни алгоритми с indexOf (при които `indexFrom` се търси на две стъпки, напр. в началото: `String startTag = "<name"; int indexFrom = xml.indexOf(startTag); indexFrom = xml.indexOf(">", indexFrom+1)`).

Заместване на подниз

С помощта на методите replace, един подниз в текущия низ се заменя с друг. В резултат се получава нов низ. Параметрите на методите са два и може да са от тип char или от тип String – търси се първият параметър и се заменя с втория. В следващия пример е показана работа с различните методи.

```
String s = "аб г д";

// заместват се всички срещания - параметри от тип char
System.out.println(s.replace(' ', 'в'));
// заместват се всички срещания - параметри от тип String
System.out.println(s.replace("аб ", "абв"));
// само първия намерен подниз се заменя
System.out.println(s.replaceFirst(" ", "в"));
// заместват се всички срещания
System.out.println(s.replaceAll(" ", "в"));

Резултат:
абвгвд
абвг д
абвг д
абвгвд
```

Разликата между методите replace и replaceAll е, че търсеният низ в replaceAll може да се описва с регулярни изрази. Първият параметър на метода replaceFirst, също може да е регулярен израз.

Регулярни изрази

Правилата за съставяне на регулярни изрази са много обширни и не са разгледани в настоящата книга. Прилагат се с помощта на класа java.util.regex.Pattern. Подробна информация за тях има на адрес:
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Някои елементарни примери за използването на регулярни изрази са показани в следващия клас:

```
public class RegularExpressionsTest {
  // Примери за регулярни изрази
  public static void main(String[] args) {
    String s = "1234 абвг";

    // заменя всички срещания на елементи от множеството [аг] с "-"
    System.out.println(s.replaceAll("[аг]", "-"));  // 1234 -бв-
    // заменя всички срещания на елемент от интервала [0-9] със "*"
    System.out.println(s.replaceAll("[0-9]", "*")); // **** абвг
    // заменя всички срещания на елементи, които не са от множеството [аг] с "-"
    System.out.println(s.replaceAll("[^аг]", "-")); //-----а--г
```

```
  }
}

Резултат:
1234 -бв-
**** абвг
-----а--г
```

Форматиране на низове

Като алтернатива на конкатенацията на низове, често използван подход е форматирането на низове. При форматирането, във форматиращ шаблонен низ се описват очаквани стойности от различни типове. Конкретните стойности се задават допълнително. Форматирането се извършва с помощта на класа java.util.Formatter. В класът String е достъпен статичен метод за форматиране. Тъй като методът е статичен, за използването му не е необходимо да се създава обект. Като първи параметър на метода се задава форматиращ низ, а следващите параметри са променлив брой (дефинирани с оператора „...”). При друг вариант на метода има добавен още един параметър в началото на списъка с параметри – от клас java.util.Locale – който се използва при форматиране на зависими от региона обекти. По аналогичен начин може да се използва метода System.out.printf().

За форматирането на низове има множество правила – те не са разгледани подробно в книгата. Очакваните параметри могат да бъдат числа, низове, булеви стойности, части от дата и време и др. Шаблоните могат да бъдат разгледани на адрес https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html. Тук са показани само елементарни примери:

```
public class StringFormat {
  public static void productInfoFormated(String product, double price){
    // %s - низ; %n - край на ред
    // %.2f - реално число с два символа след десетичната запетая
    System.out.printf("Продукт %s е с цена %.2f лв.%n", product, price);
  }
  public static void main(String[] args) {
    int i = 9;
    double d = i;

    // %d - цяло число; %o- число в осмичен вид; %f реално число; %n - край на ред
    System.out.printf("Числото %d(10) в осмичен вид %o(8) и като реално %f%n",
                      i, i, d);

    productInfoFormated("П1", 0.50);
    productInfoFormated("П2", 3);
  }
}

Резултат:
Числото 9(10) в осмичен вид 11(8) и като реално 9.000000
Продукт П1 е с цена 0.50 лв.
Продукт П2 е с цена 3.00 лв.
```

Други методи в класа String

Други методи на класа String, които могат да се използват в различни ситуации:

- ***String trim()*** – връща нов низ, при който крайните интервали и други „бели полета” на текущия низ са премахнати.
- ***char[] toCharArray()*** – връща масив от символите на низа.
- ***String[] split(String regex)*** – разделя текущия низ на части, определени от зададен като параметър разделител. Параметърът може да е регулярен израз, включващ

множество различни символи, които желаем да се разглеждат като разделител. Връща масив от низове за всяка от частите.

- ***static String join(CharSequence delimiter, CharSequence... elements)*** – метод обратен на split. CharSequence е интерфейс, който се реализира от класовете String, StringBuffer, StringBuilder и др. т.е. като параметри може да за даваме например обекти от клас String. Методът съединява елементите, които са зададени като масив или списък с произволен брой аргументи. Връща низ с всички елементи, залепени помежду си, като за разделител се използва първият параметър.

Следва пример за употреба на описаните методи:

```
public class StringOtherMethods {
  public static void main(String[] args) {
    // премахва крайните бели полета
    System.out.println(" \n \t  текст   ".trim()); // текст

    // преобразува низ в масив от символи
    char [] chars = "абв".toCharArray();
    System.out.println(Arrays.toString(chars));  // [а, б, в]

    // Разделя низ на части определени от разделител и ги записва в масив
    String [] cities = "Пловдив, София, Варна, Бургас".split(", ");
    System.out.println(Arrays.toString(cities));// [Пловдив, София, Варна, Бургас]

    // Свързва в низ градовете от масива cities с разделител "; "
    String citiesNew = String.join("; ", cities);
    System.out.println(citiesNew); // Пловдив; София; Варна; Бургас
  }
}

Резултат:
текст
[а, б, в]
[Пловдив, София, Варна, Бургас]
Пловдив; София; Варна; Бургас
```

Специфични методи за класовете StringBuffer и StringBuilder

Класовете StringBuffer и StringBuilder имат идентични помежду си методи, с еднакви имена и логика. Използват се вместо класа String с цел по-оптимално използване на паметта. Освен методи като length(), charAt(), indexOf(), lastIndexOf, substring(), присъщи на класа String, класовете **StringBuffer и StringBuilder притежават и методи със специфична логика,** породена от това, че те са променяеми:

- ***append(параметър)*** – добавя параметъра към края на текущия обект. Връща референция към текущия обект.
- ***delete(int start, int end)*** – премахва от текущия обект символите от индекс start до индекс end-1. Връща референция към текущия обект.
- ***deleteCharAt(int index)*** – премахва в текущия обект символа от зададена позиция index. Връща референция към текущия обект.
- ***insert(...)*** – методи за вмъкване на подниз, при които се задават като параметри начална позиция за вмъкване, параметри за вмъкване от различни типове и др. Връщат референция към текущия обект.
- ***reverse()*** – обръща символите на текущия обект.
- ***String toString()*** – извършва преобразуване на текущия обект до низ от тип String.

В следващия клас са дадени няколко примера с описаните методи:

```
public class StringBufferOtherMethods {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("абв");
            sb.deleteCharAt(1);      // изтриване на символ
            System.out.println(sb); // ав (извиква се метода toString)
            sb.insert(1, "Б");       // вмъкване на символ
            System.out.println(sb); // аБв
            sb.reverse();            // обръщане
            System.out.println(sb); // вБа
      }
}

Резултат:
ав
аБв
вБа
```

Въпроси и задачи за упражнения

1. Разгледайте техниките за форматиране на низове. Създайте собствени примери.
2. Разгледайте техниките за създаване на регулярни изрази. Създайте собствени примери.
3. Нека е даден масив от низове – имената на хора. Изведете
   а. хората с първо име „Иван”;
   б. хората, съдържащи низа „Иван” в името;
   в. хората, чиито презиме или фамилия завършват на „ва”;
   г. минимална, максимална и средната дължина на имената;
   д. имената с минимална дължина;
   е. масива, сортиран във възходящ лексикографски ред;
4. За даден текстов низ, отпечатайте различните думи и броя на срещанията им.
5. По зададено ЕГН изведете информация дали то е валидно, датата на раждане и пола на притежателя му.
6. Нека е даден структуриран документ от вида

```
String xml = "<products>"
          + "  <product>"
          + "    <name>Продукт 1</name>"
          + "    <price>Цена 1</price>"
          + "    <count>Брой 1</count>"
          + "  </product>"
          + "  <product>"
          + "    <name>Продукт 2</name>"
          + "    <price>Цена 2</price>"
          + "    <count>Брой 2</count>"
          + "  </product>"
          + "</products>";
```

   Изведете в конзолата списък на продуктите и съответните им цени във формат „Продукт – цена – брой”, а също и общият брой на продуктите, минимална, максимална и средно аритметична цена.
7. Даден е CSV текст от вида:

```
   String csv = "1;H;Водород;1766;Cavendish;Hydrogen\n"
              + "2;He;Хелий;1895;Ramsay;Helium\n"
              + "3;Li;Литий;1817;Arfwedson;Lithium\n"
              + "4;Be;Берилий;1797;Vauquelin;Beryllium\n"
```

```
                + "5;B;Бор;1808;Davy en Gay-Lussac;Boron\n"
                + "6;C;Въглерод;;;Carbon\n"
                + "7;N;Азот;1772;Rutherford;Nitrogen\n"
                + "...";
```

В него е записана информация за химични елементи: пореден номер, символ, име на български, година на откриване, откривател, име на английски. Генерирайте XML документ, запишете го в променлива от тип низ и го отпечатайте. Генерираният XML документ да е в следния формат:

```
<elements>
     <element>
            <number>1</number>
            <symbol>H</symbol>
            <name_bg>Водород</name_bg>
            <name_en>Hydrogen</name_en>
            <discoverer>Cavendish</discoverer>
            <year_discovered>1766</year_discovered>
     </element>
     <element>
            <number>2</number>
      ...
</elements>
```

8. Във валиден HTML документ има данни, описани в таблици (таг <table>). Изведете данните (таг <td> или <th>) за всеки ред (таг <tr>) от таблицата на самостоятелен ред в конзолата, разделени с ; (точка и запетая).

Упътвания към задачите

1. Използвайте документацията на класа java.util.Formatter, примери на адреси https://docs.oracle.com/javase/tutorial/i18n/format/index.html, http://alvinalexander.com/programming/printf-format-cheat-sheet, http://www.java2s.com/Book/Java/Essential-Classes/Formatter.htm, както и други ресурси в Интернет.
2. Използвайте документацията на класа java.util.regex.Pattern, Java tutorial за регулярни изрази, както и други ресурси в Интернет, напр., http://www.java2s.com/Book/Java/Essential-Classes/Regular_Expression_Processing.htm.
3. Примерен масив `String names = {"Иван Иванов Иванов", "Мария Иванова Петрова"...};`. Реализирайте обхождане на масива в цикъл. За всяко от имената използвайте подходящи методи на класа String: а) startsWith(); б) contains(); в) endsWith(); г) и д) length() и алгоритми за минимална, максимална и средно аритметична стойност; е) compareTo() и някой от алгоритмите за сортиране.
4. Определете всички възможни разделители между думите (бели полета, пунктуационни знаци). Намерете отделните думи (без значение от регистъра), например с методите toLowerCase и split. Използвайте ефективен начин за обработка на масива, при който извеждате думите без повторения и броя им.
5. Използвайте правилата за съставяне на ЕГН, описани например на следния адрес: https://bg.wikipedia.org/wiki/Единен_граждански_номер. Използвайте substring за извличане на отделните части на ЕГН-то.
6. Използвайте примера в част Извличане на подниз или друга техника за парсиране – регулярни изрази, XML API-та: DOM, JDOM, SAX, StAX.

7. За парсиране използвайте метода split на класа String и/или клас Scanner и/или друга техника. За генериране на документа може да използвате оператор за конкатенация или да създадете форматиращ низ за представяне на един химичен елемент. При конкатенация, с цел оптимизация, използвайте класа StringBuilder (или StringBuffer).
8. Използвайте решенията на предходните две задачи. Потърсете в Интернет примерен HTML текст, съдържащ таблици и го използвайте като входен низ. В решението обработвайте и случая, при който в таговете има и атрибути.

Използвана литература

1. Gosling, J., B. Joy, G. Steele, G. Bracha, A. Buckley, The Java® Language Specification, Java SE 8 Edition, 13.02.2015, http://docs.oracle.com/javase/specs/jls/se8/html/, последно посещение на 4.01.2016 г.
2. Java API Specifications, http://www.oracle.com/technetwork/java/api-141528.html, последно посещение на 4.01.2016 г.
3. Java Book, http://www.java2s.com/Book/Java/CatalogJava.htm, последно посещение на 4.01.2016 г.
4. Java Community, https://community.oracle.com/community/java, последно посещение на 4.01.2016 г.
5. Java Examples, http://www.java-examples.com, последно посещение на 4.01.2016 г.
6. Java keywords with examples, http://www.codejava.net/java-core/the-java-language/java-keywords, последно посещение на 4.01.2016 г.
7. Java world, http://www.javaworld.com, последно посещение на 4.01.2016 г.
8. The Java™ Tutorials, http://docs.oracle.com/javase/tutorial/, последно посещение на 4.01.2016 г.
9. Азълов, П., Ф., Златарова, С ++ в примери, задачи и приложения, Изд. Просвета – гр. София, 2011, ISBN: 978-954-01-2521-3.
10. Брус, Е., Да мислим на JAVA, том 1, Изд. СофтПрес - гр. София, 2001, ISBN: 954-685-174-4.
11. Брус, Е., Да мислим на JAVA, том 2, Изд. СофтПрес - гр. София, 2001, ISBN: 954-685-174-4.
12. Кен, А., Д. Гослинг, Д., Холмс, Програмният език JAVA, Изд. ИнфоДар – гр. София, 2001, ISBN: 954-761-034-1.
13. Кристофър, С., У., Джо, Тайните на JAVA. Програмиране в Интернет, том 1, Изд. Lio Book Publishing – гр. София, 1997.
14. Кристофър, С., У., Джо, Тайните на JAVA. Програмиране в Интернет, том 2, Изд. Lio Book Publishing – гр. София, 1997.
15. Лендерт, А., Алгоритми и структури от данни в C++, ИК Софтех – гр. София, 2001, ISBN 954-8495-25-2.
16. Наков, П., П., Добриков, Програмиране = ++ Алгоритми; Изд. TopTeam Co. – гр. София, 2002, ISBN: 954-8905-06-X, http://www.programirane.org, последно посещение на 4.01.2016 г.
17. Наков, С. и колектив, Въведение в програмирането с Java, Изд. Фабер – гр. Велико Търново, 2009, ISBN: 978-954-400-055-4, http://www.introprogramming.info/intro-java-book, последно посещение на 4.01.2016 г.
18. Рос, Д., Х., Саймън, Основи на алгоритмите, Изд. Алекс Софт – гр. София, 2006, ISBN: 954-6561-42-8.