

КРАСИМИР СТОЕВ

НИКОЛАЙ ТОМИТОВ



JAVA **за** ВСЕКИ

ИЛИ КАК ДА СТАНЕМ JAVA ПРОГРАМИСТИ

**ОСНОВИ НА ПРОГРАМИРАНЕТО
АЛГОРИТМИ ЗА СОРТИРАНЕ**

КНИГАТА ВКЛЮЧВА НАД 100 ЗАДАЧИ,
КАКТО И РЕШЕНИЯ НА НЯКОИ ОТ ТЯХ

ЧАСТ

1

Table of Contents

| | |
|---|----|
| Глава 1. Въведение в програмирането..... | 16 |
| В тази глава..... | 16 |
| Инсталация на Java Development Kit (JDK)..... | 17 |
| Инсталация на Eclipse..... | 18 |
| Компилятор, интерпретатор и виртуална машина..... | 19 |
| Езици от високо или ниско ниво..... | 19 |
| Какво е компилатор?..... | 20 |
| Какво е интерпретатор?..... | 21 |
| Какво е Java виртуална машина (JVM)?..... | 21 |
| Работа със средата за разработка Eclipse..... | 22 |
| Създаване на нов Java проект в Eclipse..... | 23 |
| Създаване на нов клас в даден проект..... | 24 |
| Създаване и пускане на първата Java програма..... | 25 |
| Повече за класовете и файловете..... | 26 |
| Упражнения..... | 27 |
| Глава 2. Примитивни типове данни, променливи и оператори..... | 28 |
| В тази глава..... | 28 |
| Какво са променливите?..... | 29 |
| Типове данни в езика Java..... | 30 |
| Примитивни типове данни..... | 30 |

| | |
|---|----|
| Целочислени типове данни..... | 30 |
| Типове данни за числа с плаваща запетая..... | 31 |
| Булев тип..... | 31 |
| Знаков тип..... | 32 |
| Референтни типове данни..... | 32 |
| Работа с променливи в езика Java..... | 32 |
| Създаване на променлива от даден тип..... | 32 |
| Наименоване на променливите в Java..... | 33 |
| Стойности по подразбиране на променливите в Java..... | 34 |
| Литерали в езика Java..... | 35 |
| Целочислени литерали..... | 35 |
| Литерали с плаваща запетая..... | 35 |
| Знакови литерали..... | 35 |
| Операции с променливи..... | 36 |
| Извеждане на променливи в конзолата..... | 36 |
| Четене на променливи от конзолата..... | 40 |
| Аритметични оператори..... | 42 |
| Оператори за сравнение..... | 43 |
| Логически оператори..... | 45 |
| Оператори за присвояване..... | 46 |
| Оператори за увеличаване и намаляване с единица..... | 46 |
| Упражнения..... | 47 |

Глава 1. Въведение в програмирането

| | |
|--|----|
| Глава 3. Бройни системи. Побитови оператори..... | 49 |
| В тази глава..... | 49 |
| Дефиниция на бройна система..... | 50 |
| Десетична бройна система (DECIMAL)..... | 50 |
| Двоична бройна система (BINARY)..... | 51 |
| Защо ни е нужно да имаме такава бройна система ?..... | 51 |
| Как броим в двоичната бройна система ?..... | 51 |
| Шестнадесетична бройна система (HEXADECIMAL)..... | 53 |
| Представяне на числа в различна бройна система в Java..... | 53 |
| Преобразуване на числата от една бройна система в друга..... | 54 |
| Преобразуване от десетична в двоична бройна система..... | 55 |
| Преобразуване от двоична в десетична бройна система..... | 56 |
| Операции с двоични числа. Побитови оператори..... | 58 |
| Оператор за побитово отрицание (NOT)..... | 58 |
| Оператор за побитово И (AND)..... | 59 |
| Оператор за побитово ИЛИ (OR)..... | 60 |
| Оператор за изключващо ИЛИ (XOR)..... | 61 |
| Побитови измествания..... | 62 |
| Упражнения..... | 64 |
| Глава 4. Условни оператори..... | 65 |
| В тази глава..... | 65 |
| Условен оператор if..... | 66 |

| | |
|--|-----|
| Оператор if-else..... | 67 |
| Вложени if оператори..... | 69 |
| Оператор switch-case..... | 70 |
| Оператор за сравнение ?:..... | 74 |
| Комбиниране на няколко условия, използвайки логически оператори..... | 76 |
| Упражнения..... | 78 |
| Глава 5. Цикли..... | 79 |
| В тази глава..... | 79 |
| Дефиниция за цикъл..... | 80 |
| Цикъл „while“..... | 82 |
| Цикъл „do-while“..... | 86 |
| Цикъл „for“..... | 88 |
| „For“ цикъл с няколко управляващи променливи..... | 91 |
| Безкраен „For“ цикъл..... | 92 |
| Вложени цикли..... | 92 |
| Ключова дума „break“..... | 95 |
| Ключова дума „continue“..... | 96 |
| Упражнения..... | 97 |
| Глава 6. Масиви..... | 98 |
| В тази глава..... | 98 |
| Защо се нуждаем от масиви и какво представляват?..... | 99 |
| Декларация и инициализация на масив..... | 100 |

Глава 1. Въведение в програмирането

| | |
|--|-----|
| Достъп и обхождане на елементите на масив..... | 101 |
| Обхождане на елементите на масив..... | 102 |
| Четене и отпечатване на масив в конзолата..... | 103 |
| Копиране и сравняване на масиви..... | 104 |
| Разликата между Стек (Stack) и Хийп (Heap)..... | 104 |
| Как работят операторите "=" и "==" при референтни типове данни?.... | 105 |
| Упражнения..... | 108 |
| Упътвания..... | 109 |
| Глава 7. Двумерни масиви..... | 110 |
| В тази глава..... | 110 |
| Какво са двумерните масиви..... | 111 |
| Връзка с едномерните масиви..... | 111 |
| Създаване и инициализация..... | 112 |
| Достъп до елементите на двумерен масив..... | 113 |
| Обхождане, четене и извеждане на двумерни масиви..... | 113 |
| Сума на елементите, минимален и максимален елемент в двумерен масив | 114 |
| Упражнения..... | 116 |
| Упътвания..... | 116 |
| Глава 8. Методи. Рекурсия..... | 118 |
| В тази глава..... | 118 |
| Защо повтарянето на код е лоша практика..... | 119 |

| | |
|---|-----|
| Въведение в методите..... | 120 |
| Декларация и имплементация на метод..... | 121 |
| Връщане на стойност от метод..... | 123 |
| Извикване на метод..... | 125 |
| Пример за програма с методи..... | 128 |
| Област на видимост на променливите..... | 131 |
| Предаване на параметри към метод..... | 132 |
| Рекурсия..... | 138 |
| Видове рекурсия..... | 138 |
| Задължителни елементи на рекурсията..... | 139 |
| Стъпка на рекурсията..... | 139 |
| Дъно на рекурсията..... | 139 |
| Безкрайна рекурсия..... | 140 |
| Добри практики при рекурсията..... | 140 |
| Рекурсия и итерация..... | 141 |
| Упражнения..... | 141 |
| Глава 9. Символни низове..... | 143 |
| В тази глава..... | 143 |
| Въведение в низовете..... | 144 |
| Декларация, инициализация, вход и изход от конзолата..... | 144 |
| Основни операции с текстови низове..... | 145 |
| Дължина на низ..... | 145 |

Глава 1. Въведение в програмирането

| | |
|---|-----|
| Достъпване и обхождане символите на даден низ..... | 145 |
| Пример: брой думи в даден текст..... | 146 |
| Слепване (конкатенация) на низове..... | 147 |
| Представяне на низовете в паметта..... | 148 |
| Пул(pool) от обекти..... | 148 |
| Сравнение на низове..... | 149 |
| Други начини за сравнение на низове..... | 151 |
| equalsIgnoreCase()..... | 151 |
| compareTo()..... | 151 |
| Допълнителни операции върху низове..... | 152 |
| split()..... | 152 |
| join()..... | 152 |
| substring()..... | 153 |
| replace()..... | 153 |
| trim()..... | 154 |
| indexOf()..... | 154 |
| Методите toLowerCase() и toUpperCase()..... | 155 |
| Форматиране на низове с метода String.format()..... | 155 |
| contains()..... | 156 |
| Класът StringBuilder – основни операции и употреба..... | 157 |
| Деклариране и инициализация..... | 157 |

| | |
|--|-----|
| Модификация на StringBuilder обекти – методите insert, delete, replace и append..... | 158 |
| insert()..... | 158 |
| delete()..... | 158 |
| replace()..... | 158 |
| append()..... | 159 |
| Сравняване на стойности от тип StringBuilder..... | 159 |
| Сравнение с класа String..... | 160 |
| Упражнения..... | 160 |
| Упътвания..... | 161 |
| Глава 10. Въведение в алгоритмите..... | 163 |
| В тази глава..... | 163 |
| Какво е алгоритъм..... | 163 |
| Защо е нужно да изучаваме алгоритми..... | 164 |
| Как да измислим алгоритъм ?..... | 165 |
| Сложност на алгоритми – опростено обяснение..... | 167 |
| Сложност по време..... | 168 |
| Сложност по памет..... | 168 |
| Видове сложност на алгоритми..... | 169 |
| Big O нотация..... | 169 |
| Константна сложност – $O(1)$ | 169 |
| Логаритмична сложност – $O(\log N)$ | 170 |

Глава 1. Въведение в програмирането

| | |
|---|-----|
| Линейна сложност – $O(N)$ | 170 |
| Енлог сложност – $O(N\log N)$ | 171 |
| Квадратична сложност – $O(N^2)$ | 171 |
| Кубична сложност – $O(N^3)$ | 172 |
| Експоненциална сложност - $O(2^N)$ | 172 |
| Факториелна сложност – $O(N!)$ | 172 |
| Сложност в различни ситуации..... | 172 |
| Сложност при най-благоприятни условия - Best case complexity..... | 173 |
| Сложност при най-неблагоприятни условия - Worst case complexity. . | 173 |
| Сложност в общия случай - Average case complexity..... | 173 |
| Глава 11. Сортиране - метод на мехурчето и на пряката селекция..... | 174 |
| В тази глава..... | 174 |
| Метод на мехурчето – основна идея..... | 175 |
| Имплементация на алгоритъма..... | 176 |
| Оптимизиране на алгоритъма..... | 177 |
| Анализ на алгоритъма - сложност по време и по памет..... | 179 |
| Метод на пряката селекция – основна идея..... | 180 |
| Имплементация на алгоритъма..... | 180 |
| Оптимизации на алгоритъма..... | 182 |
| Анализ на алгоритъма - сложност по време и по памет..... | 184 |
| Глава 12. Сортиране чрез броене. Radix сортиране..... | 185 |
| В тази глава..... | 185 |

| | |
|--|-----|
| Сортиране чрез броене – основна идея..... | 186 |
| Имплементация на алгоритъма..... | 189 |
| Анализ на алгоритъма..... | 191 |
| Radix сортиране – основна идея..... | 192 |
| Стабилни и нестабилни сортиращи алгоритми..... | 193 |
| Имплементация на алгоритъма..... | 194 |
| Анализ на алгоритъма..... | 197 |
| Глава 13. Пирамидално сортиране (HeapSort)..... | 199 |
| В тази глава..... | 199 |
| Въведение в пирамидата като структура от данни..... | 200 |
| Пирамидално условие..... | 201 |
| Реализация на пирамидата с масив..... | 201 |
| Операции с пирамида..... | 203 |
| Вмъкване на елемент..... | 203 |
| Изтриване на елемент..... | 204 |
| Създаване на пирамида от произволен масив..... | 205 |
| Пирамидално сортиране – основна идея..... | 208 |
| Имплементация на алгоритъма..... | 210 |
| Анализ на алгоритъма - сложност по време и по памет..... | 212 |
| Глава 14. Сортиране чрез сливане. Бинарно търсене..... | 214 |
| В тази глава..... | 214 |
| Алгоритми от типа „Разделяй и владей“ | 215 |

Глава 1. Въведение в програмирането

| | |
|--|-----|
| Сортиране чрез сливане – основна идея..... | 216 |
| Имплементация в Java..... | 219 |
| Анализ на алгоритъма - сложност по време и по памет..... | 221 |
| Двоично търсене – основна идея..... | 221 |
| Имплементация в Java..... | 224 |
| Анализ на алгоритъма..... | 225 |
| Глава 15. Бързо Сортиране..... | 227 |
| В тази глава..... | 227 |
| Основна идея на алгоритъма..... | 228 |
| Имплементация в Java..... | 230 |
| Анализ на алгоритъма..... | 233 |
| Оптимизации на алгоритъма..... | 234 |
| Глава 16. Примерни задачи..... | 236 |
| В тази глава..... | 236 |
| Задачи за променливи и типове данни..... | 237 |
| Задачи за бройни системи и побитови оператори..... | 240 |
| Задачи за условни оператори..... | 243 |
| Задачи за цикли..... | 245 |
| Задачи за едномерни масиви..... | 248 |
| Задачи за двумерни масиви..... | 252 |
| Задачи за методи..... | 254 |
| Задачи за рекурсия..... | 255 |

| | |
|---|-----|
| Задачи за символни низове..... | 256 |
| Задачи за сортиращи алгоритми и двоично търсене..... | 260 |
| Глава 17. Решения..... | 264 |
| В тази глава..... | 264 |
| Увод..... | 265 |
| Как да решаваме задачи по програмиране..... | 265 |
| Решения на задачи за променливи и типове данни..... | 265 |
| Решения на задачи за бройни системи и побитови оператори..... | 269 |
| Решения на задачи за условни оператори..... | 276 |
| Решения на задачи за цикли..... | 286 |
| Решения на задачи за едномерни масиви..... | 293 |
| Решения на задачи за двумерни масиви..... | 301 |
| Решения на задачи за методи..... | 306 |
| Решения на задачи за рекурсия..... | 310 |
| Решения на задачи за символни низове..... | 315 |
| Решения на задачи за сортиращи алгоритми и двоично търсене..... | 321 |

Глава 1. Въведение в програмирането

В тази глава

Настройка на работната среда

Компилятор, интерпретатор и виртуална машина


Работа с Eclipse

Инсталация на Java Development Kit (JDK)

За да програмирате на Java, едно от нещата, които са ви необходими, е Java Development Kit – набор от стандартни библиотеки и програмни средства за изпълнение на вашите програми. По времето, когато тази книга е писана, последната версия на Java, а от там и на JDK е версия 8. Разбира се, за нуждите на тази книга по-стари и по-нови версии също биха свършили работа. Можете да свалите JDK 8 от следния уеб адрес :

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>


В секцията "Java SE Development Kit 8u31" изберете "Accept License Agreement" бутона преди да продължите и свалете подходящата версия на JDK спрямо вашата операционна система и архитектура.



Уверете се, че сте избрали правилната версия спрямо вашата операционна система и архитектура. Например 64 битова версия на JDK няма да върви под 32 битова операционна система. За потребителите на Windows 7 можете да проверите колко битова е вашата операционна система натискайки Start бутона -> Control Panel → System. В излезлия прозорец под раздела System ще видите System type :

System

Rating:

 Your Windows Experience Index needs to be refreshed

Processor:

Intel(R) Celeron(R) CPU G1610 @ 2.60GHz 2.60 GHz

Installed memory (RAM):

8.00 GB (7.70 GB usable)

System type:

64-bit Operating System

Pen and Touch:

No Pen or Touch Input is available for this Display

Изтеглете версията за "Windows x86" ако вашата система е 32 битова или "Windows x64" в противен случай.

След като свалите необходимия файл, инсталирайте JDK – не би трябвало да имате особени проблеми, тъй като инсталацията е направена

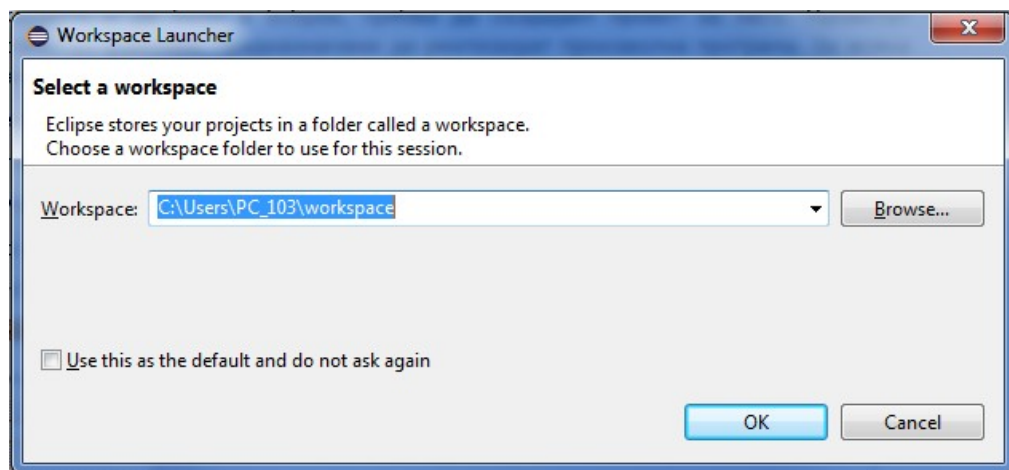
максимално лесна и удобна. Имайте предвид, че може да се наложи да рестартирате компютъра си, за да се отразят промените.

Инсталация на Eclipse

Докато JDK ни позволява да изпълняваме и да разработваме нашите Java програми, то Eclipse ни дава подходяща среда за разработка. Можем да пишем Java код дори и в Notepad, но това, което ни дава Eclipse, е улеснение при работата, специално с Java код. Можете да свалите Eclipse от следния уеб адрес :

<https://eclipse.org/downloads/>

Отново изберете вашата операционна система и свалете 32 или 64 битов Eclipse. Не е нужна инсталация, просто разархивирайте сваления архив някъде, за предпочитане на C:\ или D:\ ако използвате Windows. В разархивираната вече папка Eclipse може да стартирате eclipse.exe и ако всичко е наред, ще бъдете помолени да въведете къде да бъде работното ви пространство (workspace) :



Изберете произволна папка и натиснете бутон ОК. В избраната директория ще бъде създадена служебна информация от Eclipse за вашето работно пространство и в последствие там ще създадем първия ни Java проект. След избора на **workspace** ще видим началния екран на Eclipse, който можем да затворим с "X" бутона непосредствено до "Welcome".

Компилатор, интерпретатор и виртуална машина

Преди да продължим нататък, е важно да се запознаем с няколко основни термина, които ще ни помогнат да разберем как точно функционира езика Java и какво е специфично при програмите, написани на този език.

Езици от високо или ниско ниво

При езиците от високо ниво кода е по-лесно разбираем за програмиста. Можем с по-малко код да постигнем повече функционалност и разполагаме с голям брой предварително готови библиотеки и инструменти, с които да работим. От друга страна, при езиците от ниско ниво, кода е неразбираем за програмиста, много дълъг, необходими са много инструкции, за да постигнем нещо малко, но пък за сметка на това кода се изпълнява доста по-бързо. Причината за това е, че при езиците от ниско ниво ние пишем кода така, че той да е специфичен за конкретната компютърна архитектура (процесор, памет, операционна система и т.н.), върху която ще се изпълнява. С други думи – програмите от ниско ниво са максимално лесни за изпълнение от процесора. От там идва и друг недостатък на програма, писана на тези езици, а именно, че трябва да се преправя преди да се пуска на различни машини.

Пример за език от ниско ниво (Асемблер) :

```
.or $300  
main    ldy #$00  
.1      lda str,y
```

```

        beq .2
        jsr $fded
        iny
        bne .1
.2      rts
str     .as "HELLO WORLD"
        .hs 0D00

```

Пример за език от високо ниво (Java) :

```

public class Demo {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}

```

Какво е компилатор?

Компилатор е програма, която преобразува код, написан на език от високо ниво (Java, C, C++, C#) към език или формат от ниско ниво, разбираем и изпълним от машината.

Например, когато компилираме програма, написана на езика C, получаваме изпълним .exe файл под Windows, състоящо се от код на машинен език, който след това можем да изпълним. Компилацията на Java програми е малко по-сложен процес, защото води до получаването на **байт-код**. **Байт-кодът** можем да изпълним, както и ще видим по-нататък, използвайки Java виртуална машина (JVM).

Какво е интерпретатор?

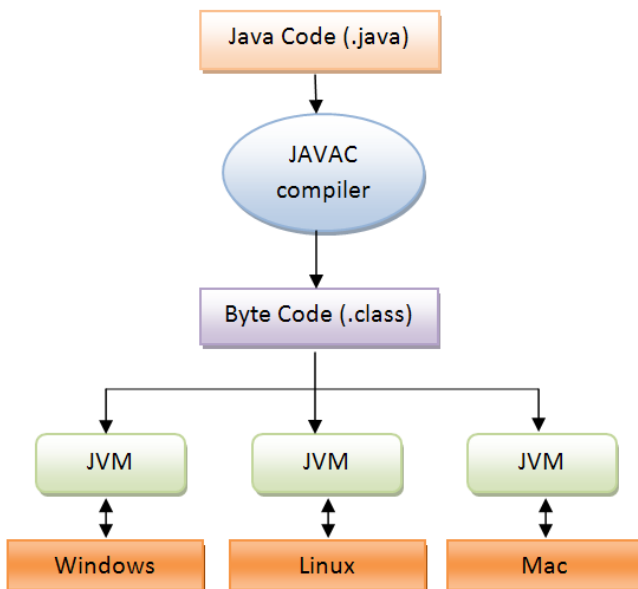
Интерпретатор е програма, която директно изпълнява код, написан на някакъв програмен или скриптов език, без преди това да го компилира към друг език от по-ниско ниво. Примери за езици, които се изпълняват от интерпретатори са JavaScript, който се изпълнява от Web Browser-а на вашия компютър, PHP, Python, Ruby и т.н.

Какво е Java виртуална машина (JVM)?

Java виртуална машина е програма, която изпълнява код, получен преди това чрез компилация от език от високо ниво. Такъв код ще наричаме **байт-код**. Байт-кодът е още един, междинен формат заедно с кода, който програмиста пише и машинния език, който компютъра изпълнява. В езика Java кодът, който пишем, се компилира до байт-код и впоследствие се изпълнява от Java виртуална машина. Какви са предимствата на този подход?

- лесната преносимост между различните платформи - веднъж написана и компилирана до байт-код, една Java програма може да бъде стартирана на всякакви устройства, независимо от архитектурата или операционната им система. Единственото условие е тези устройства да имат инсталирана виртуална машина.
- виртуалната машина освобождава сама заетата и ненужна вече памет от нашата програма.
- висока степен на сигурност, тъй като програмите се изпълняват от виртуалната машина в точно определена рамка, като RAM памет и дисково пространство.

По-долу нагледно можем да видим процеса по компилиране и изпълнение на една Java



Като обобщение на казаното до момента - за да изпълнявате готови Java програми, писани от вас или някой друг, необходимото условие е да имате инсталирано JRE, което включва Java виртуална машина и няколко стандартни библиотеки за работа. За да разработвате, компилирате и изпълнявате ваши собствени Java програми, ви е необходимо да имате инсталирано JDK.

Работа със средата за разработка Eclipse

Може би една от най-разпространените среди за разработка, при това не само за Java програми, е Eclipse. Разбира се има и други популярни среди за писане на Java като NetBeans (<https://netbeans.org/>), IntelliJ IDEA (<https://www.jetbrains.com/idea/>), JCreator (<http://www.jcreator.com/>) и други. Всички тези програми не компилират Java код и не изпълняват

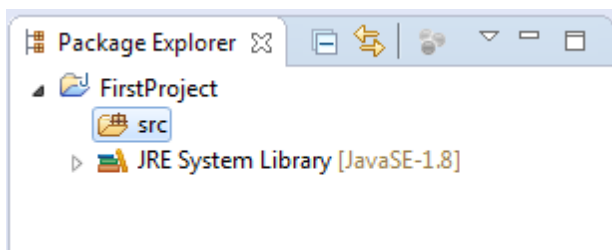
байт-кода самостоятелно, а използват JDK за тази цел. Това обаче, което те ни дават, е среда, в която лесно можем да пишем и изпълняваме дори големи Java проекти, съставени от много файлове с код и ресурси. Тези програми носят наименованието **интегрирани среди за разработка** (Integrated Development Environment или накратко IDE).

Създаване на нов Java проект в Eclipse

За да можем да пишем код, ни е необходим **клас**, който трябва от своя страна да се намира в някакъв **проект**. Проектът не е нищо повече от логическо групиране на множество файлове и папки, които ще се използват в хода на изпълнението на дадена програма. Създаването на проект в Eclipse е тривиална задача. Необходимо е да изпълним следните стъпки :

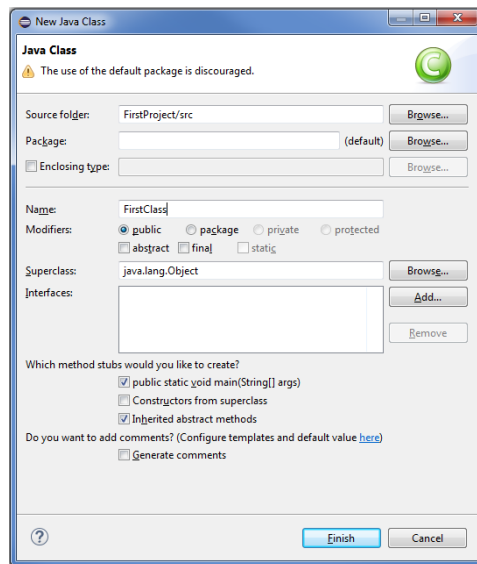
- Избираме File → New → Java Project
- Задаваме име на проекта в полето "Project name" (примерно FirstProject)
- Натискаме Finish бутона

Ако всичко е минало както трябва, би следвало да видите в левия край създаден вашият първи проект :



Създаване на нов клас в даден проект

Следващата стъпка е да създадем нов клас в нашия проект, в който ще пишем някакъв код. Класовете представляват най-общо казано логическо групиране на данни и функционалност, която да работи върху тези данни. Програмата започва да се изпълнява от клас, обозначен като Main Class и ние по-долу ще го отбележим като такъв. За да създадем нов клас натискаме с дясно копче на мишката върху "src" папката в нашия проект и избираме New → Class. Задаваме име на нашият първи клас в полето "Name" и се уверяваме, че сме избрали отметката "public static void main(String[] args)". По-долу може да видим как би трябвало да изглежда този екран :



Натискаме "Finish" и би трябвало да видим нашият нов клас, заедно с малко код в него :



```
FirstClass.java
1
2 public class FirstClass {
3
4     public static void main(String[] args) {
5         // TODO Auto-generated method stub
6
7     }
8
9 }
10
```

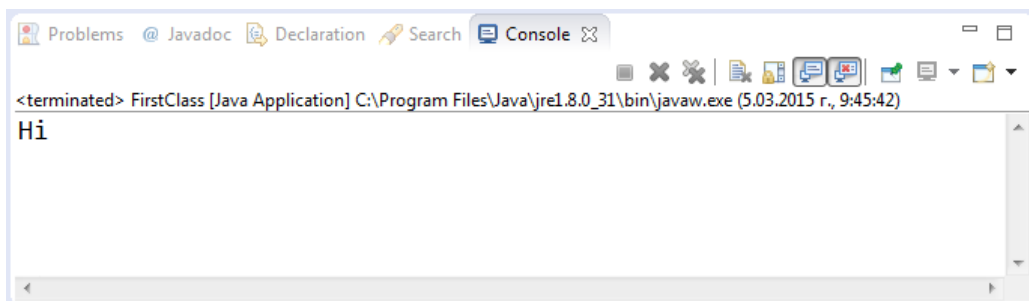
Създаване и пускане на първата Java програма

Избираме полето за писане и точно под реда, оцветен със зелено, вмъкваме следното :

```
System.out.println("Hi");
```

Този ред код инструктира Java виртуалната машина (JVM) да изведе съобщението, оградено в двойни кавички ("Hi"), в конзолата, която се намира малко по-долу. За момента може да приемем, че конзолата е просто лесно и удобно средство за четене и изкарване на данни. Можем да използваме тази конструкция повече от веднъж, за да изписваме повече от едно съобщение в конзолата. Забележете, че всяка такава инструкция завършва с ";".

За да стартираме програмата използваме  бутона, който се намира в лентата с бутони и стартираме програмата. Други варианти са да натиснем клавишите Ctrl и F11 или да изберем Run As → Run As → Java Application. Ако всичко е наред, по-долу трябва да ни се появи конзолата на Eclipse и да видим следното :



Java е език, в който имат значение малките и големите букви. Ако имате проблем със стартирането на горната програма уверете се, че сте написали всичко точно както трябва, с всички необходими препинателни знаци и спазвайки малки и големи букви. Също така, ако не получавате грешка, но не виждате конзолата, можете да я покажете на екрана, използвайки Window → Show View → Console.

Повече за класовете и файловете

Ако се загледаме в папката "src" в ляво, ще видим, че в нея има файл "FirstClass.java" (или друго име, в зависимост от това как сме наименовали нашият клас). В езика Java, кодът на всеки клас стои в отделен ".java" файл. Когато Java компилаторът компилира нашият код, наречен още **сурс-код (source code)**, той генерира като резултат файл със същото име, но с разширение ".class". Например, сурс-кодът на нашият клас FirstClass стои във файл с име FirstClass.java и компилаторът ще произведе файл с байт-код, който се казва FirstClass.class. Накрая Java виртуалната машина ще изпълни този байт-код, характерно за нашата архитектура и ние ще видим резултата в конзолата.

Упражнения

1. Модифицирайте програмата, така че да извежда надписа "Hello World!".
2. Изведете две съобщения в конзолата "Hello Java!" и под него "This is another line".
3. Изведете числата от 1 до 5 на отделни редове в конзолата.
4. Създайте нов проект, с нов клас и изведете числата от 1 до 5, но този път на един и същ ред, разделени с интервал.
5. Модифицирайте предишната задача, така че да извежда три такива реда.

Глава 2. Примитивни типове данни, променливи и оператори

В тази глава

Какво са променливите

Типове данни в езика Java

Работа с променливи и основни оператори

Какво са променливите?

Можем да си представим променливата като кутия, в която се слага нещо. В зависимост от това колко е голяма кутията, може да слагаме различни по големина неща, но не по-големи от **размера** ѝ. Също така имаме различни видове кутии – за инструменти, за обувки, за детски играчки и т.н. и не можем в кутия за инструменти да сложим обувки например, защото **типа** на кутията е различен.

Ние можем да погледнем какво има вътре в някаква кутия или с други думи да **прочетем стойността** в кутията. Понякога бихме искали да премахнем съдържанието на кутията и да го заменим с друго или с други думи – да **запишем нова стойност**. Още една важна характеристика на кутиите, е че те са надписани, и ние по надписа или по **името** на кутията можем да идентифицираме, понякога без дори да поглеждаме съдържанието вътре, какво има в тази кутия и какви неща се съхраняват в нея. Принципите и характеристиките, които показахме току що важат в пълна сила и за променливите в езиките за програмиране. Къде се съхраняват променливите? Във всички програмни езици те стоят в оперативната памет на компютъра (RAM), като в зависимост от типа си те заемат точно определено количество памет, обикновено изразено в брой битове. За какво използваме променливите? Обикновено, за да запазим някакви данни, прочетени от потребителя, да извършим някакви пресмятания с тях, да предаваме данни между различни части на нашата програма и т.н. Преди да видим обаче как можем да създаваме и използваме променливи в езика Java, ще ни е необходимо първо да се запознаем с това какви **типове данни** съществуват в езика.

Типове данни в езика Java

За да дефинираме променлива, е необходимо да укажем типа ѝ. Според типа на променливата Java виртуалната машина може да задели необходимото количество памет, която в следствие да се асоциира с името на тази променливата. Съществуват основно два типа данни : **примитивни** и **референтни**.

Примитивни типове данни

Дефинирани в езика, примитивните типове данни биват :

- целочислени типове данни – byte, short, int, long
- типове данни за числа с плаваща запетая - float, double
- булев тип – boolean
- знаков тип – char

Целочислени типове данни

Съхраняват цели числа : -2, 1, 6, 100, 0 и т.н. В зависимост от конкретния тип, позволяват съхранението на различен интервал от стойности.

По-долу можем да видим таблица с целочислените типове данни и детайли за стойностите, които можем да съхраним в променлива от такъв тип :

| Тип на данните | Размер в битове | Интервал от стойности |
|----------------|-----------------|---|
| byte | 8 | -128...127 |
| short | 16 | -32 768...32 767 |
| int | 32 | - 2,147,483,648...2,147,483,647 |
| long | 64 | -2 ⁶³ ...2 ⁶³ - 1 |

Защо са ни необходими четири различни типа, при това все за цели числа? В зависимост от броя битове, които заемат в паметта, променливите от даден целочислен тип могат да съхраняват определен интервал от стойности. Когато искаме да пестим памет, а и данните, които съхраняваме са в малък обхват (примерно оценки на ученици, вариращи от 2-ка до 6-ца), бихме предпочели "по-малък" тип данни като byte или short. Обратно, ако искаме да съхраним броя жители за дадена държава или континент ще трябва да изберем тип данни int или дори long.

Типове данни за числа с плаваща запетая

Променливите от тези типове съхраняват реални числа - 5.2, -1.3, 0.01 и т.н. Имаме два варианта, отново в зависимост от това колко големи числа ще съхраняваме, а и с каква точност след десетичната запетая :

| Тип на данните | Размер в битове | Интервал от стойности |
|----------------|-----------------|-----------------------|
| float | 32 | -3.4E+38...3.4E+38 |
| double | 64 | -1.7E+308...1.7E+308 |

Булев тип

Променлива от булев тип се означава с ключовата дума „**boolean**“. Променливите от булев тип съхраняват само две възможни стойности – истина (**true**) и лъжа (**false**) и заемат само 1 бит памет. Използват се при логически операции, като например извличане на стойността на израза „навън студено ли е и минало ли е 8 часа“. Резултатът от последния израз има стойност **истина** или **лъжа** и се съхранява в променлива от булев тип. Тези променливи се комбинират и се променят, използвайки **логически оператори**, за които ще стане въпрос по-нататък в настоящата глава.

Знаков тип

Променлива от знаков тип се означава с ключовата дума „**char**“. Променливите от знаков тип съхраняват точно един знак – като например 'a', '\$', '*' и т.н. Забележете, че стойностите се задават оградени в единични кавички. В действителност стойността на този тип променливи се пази в паметта като 16-битово положително число. Тази стойност обозначава номер на символ, който стои в кодовата таблица **Unicode (UTF-16)**. Това е таблица, съхраняваща списък с всички човешки символи и техния идентификационен номер. Когато трябва да се изведе символ на екрана, се взима стойността на променливата, търси се

символа с този идентификационен номер в кодовата таблица и се извежда.

Референтни типове данни

Стойността, която пазят променливите от този тип е "адрес на някакви данни в паметта". Тоест те не съхраняват директно някаква стойност като число или символ, а по-скоро са указатели(референции) към нещо по-голямо, някъде в паметта. Използваме референтните типове, за да пазим адреси в паметта към **масиви**, за които ще говорим малко по-късно и към **обекти**, които са извън темата на настоящата книга.

Работа с променливи в езика Java

След като се запознахме с това какво представляват променливите и основните типове данни в езика, нека сега да видим как можем да създадем променлива от даден тип, да присвоим стойност, да модифицираме тази стойност и т.н.

Създаване на променлива от даден тип

Конструкцията за създаване на променлива е следната :

`<тип на променлива> <име на променлива> [=<начална стойност>];`

Началната стойност не е задължителна, но ако не зададем стойност на променливата, то ще ѝ бъде дадена стойност по подразбиране, за което ще говорим по-късно. По принцип е добра практика винаги да даваме подходящи начални стойности на нашите променливи.

Ето няколко примера :

```
public static void main(String[] args) {  
    byte score = 99;  
    int age = 19;  
    long egn = 8181818181L;  
    double liters = 2.0;  
    float money = 50.25F;  
    boolean isLate = false;  
    char letter = 'a';  
    int noDefaultValue;  
}
```

Забележете символите "L" и "F" след променливите за ЕГН и пари. По този начин обозначаваме числовите литерали съответно като **long** и **float** числа. За литералите в езика ще поговорим по-късно в същата глава.

Наименоване на променливите в Java

Добра практика е променливите да се наименоват на английски, започвайки с малка латинска буква. Ако в името на променливата участват повече от една дума, то е прието всяка следваща дума да започва с главна буква. Интервали не се допускат в имената на променливите, както и специални символи (\$,@,* и т.н.). Като последно правило трябва да се стремим имената на променливите да са достатъчно описателни за това каква стойност пази променливата вътре в себе си. Например променливите **age**, **year** и **monthlySalary** носят информация, за съдържанието, което съхраняват, докато променливите **a**, **x**, **t**, **y1** и **c4** – не носят информация какво се пази в тях на хората, които четат вашия код.

Стойности по подразбиране на променливите в Java

Ако създадете променлива без да ѝ задавате начална стойност, то Java виртуалната машина ще зададе стойност по подразбиране на тази променлива. Можете да видите стойностите по подразбиране за различните видове променливи в таблицата по-долу :

| Тип на данните | Стойност по подразбиране |
|----------------|------------------------------|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| boolean | false |
| char | '\u0000' (или нулевият знак) |
| референтен тип | null (празен указател) |



Ако се опитате да изведете на екрана променлива, дефинирана в `main`, на която не сте задали стойност, ще получите съобщение за грешка. Затова е хубаво винаги да задавате начална стойност на променливите, веднага щом ги създадете.

Литерали в езика Java

Литералите представляват стойностите, които задаваме на нашите променливи. Ще разгледаме специално особеностите на няколко вида литерали – целочислени, с плаваща запетая и знакови.

Целочислени литерали

Ако числата, които задаваме като стойност на някаква променлива завършват с "L", то типа на литерала е **long**, в противен случай типа е **int**. Това е важно, когато задаваме стойност на променлива, която е много

голямо число. Можем да зададем стойността на дадена целочислена променлива и в различна бройна система, както ще видим в следващата глава.

Литерали с плаваща запетая

Ако зададем число с плаваща запетая като стойност, то типа по подразбиране на това число е **double**. Ако искаме да получим литерал от тип **float** трябва да добавим **"F"** накрая на числото. Това е причината, поради която, създавайки променлива от тип **float** и задавайки ѝ стойност, получаваме грешка, защото не може литерал от тип **double** да се помести в променлива от тип **float**. Решението е да задаваме стойности като **float** литерали на **float** променливи.

Знакови литерали

Задаването на стойност на променлива от знаков тип става, както видяхме по-горе, като се зададе символ, ограден в единични кавички. Какво ще стане ако искаме да зададем стойност - знака за край на ред? За тази цел в езика за измислени литералите за специалните знаци, наречени изключващи последователности ("escape sequences"), които се задават, започвайки с наклонена черта **"\"**. Можете да видите най-често използваните специални знаци в таблицата по-долу :

| Последователност | Значение |
|------------------|--------------------|
| "\n" | Нов ред |
| "\t" | Табулация |
| "\b" | Клавишът Backspace |
| "\"" | Двойна кавичка |
| "\"" | Единична кавичка |
| "\\" | Наклонена черта |

Операции с променливи

Видяхме вече как можем да създаваме променливи, които да съхраняват в паметта дадена стойност от определен тип данни. Какво можем да

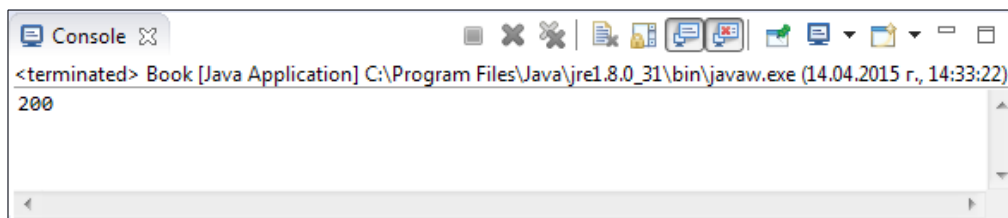
правим с нашите променливи? Можем да проверяваме какви са техните стойности, използвайки конзолата или иначе казано можем да ги извеждаме в конзолата. Също така началната стойност на нашите променливи може да се въвежда от хората, които използват нашата програма, отново през конзолата. С целочислените променливи и с тези с плаваща запетая можем да извършваме **аритметични операции**. С променливите от булев тип съответно можем да извършваме **логически операции**. Не на последно място, можем винаги да сменим началната стойност, която сме задали на дадена променлива, използвайки различните **оператори за присвояване**.

Извеждане на променливи в конзолата

Използвайки конструкцията **System.out.print()**, която видяхме в предишната глава, можем да изведем стойностите на променливите в конзолата :

```
public static void main(String[] args) {  
    int balance = 200;  
    System.out.print(balance);  
}
```

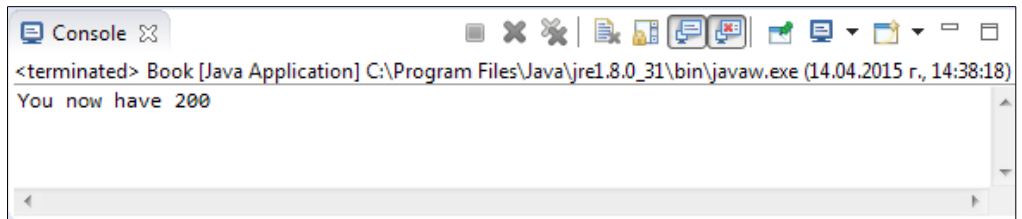
Като изход в конзолата ще видим следния резултат:



Ако искаме да изведем текстово съобщение преди стойността на дадена променлива, е необходимо да използваме знака „+“ :

```
public static void main(String[] args) {  
    int balance = 200;  
    System.out.print("You now have " + balance);  
}
```

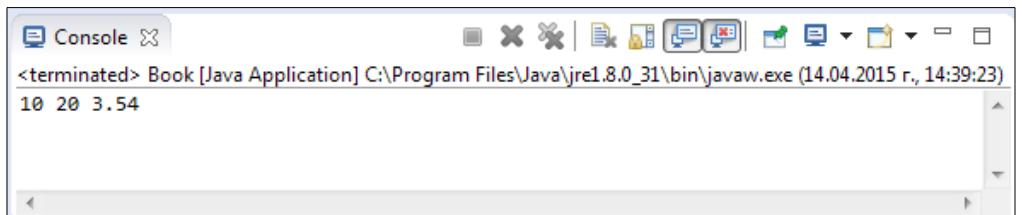
Резултатът е:



Можем да изведем на един ред стойностите на няколко променливи, разделени със запетая, използвайки следното :

```
public static void main(String[] args) {  
    int x = 10;  
    long y = 20;  
    float z = 3.54f;  
  
    System.out.print(x + " " + y + " " + z);  
}
```

Резултатът е:



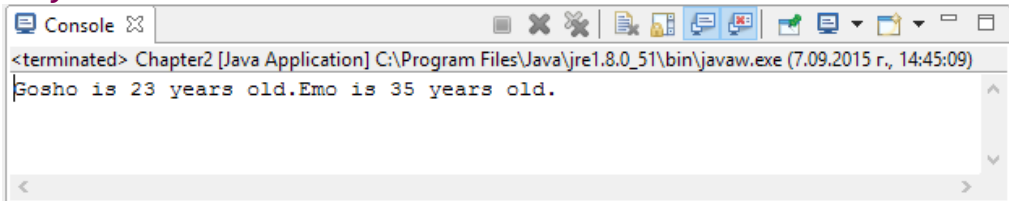
Съществува и друга конструкция за извеждане на текст в конзолата и тя е **System.out.println()**. Разликата със **System.out.print()** е, че след изписване на екрана на стойността, следващата изписана стойност ще бъде на нов ред. Дефакто **println** означава **print-line**. Отпечатва на екрана някаква стойност, след което преминава на нов ред.

Нека дадем пример с изписването на две стойности чрез **System.out.print()** и чрез **System.out.println()**.

```
public static void main(String[] args) {  
  
    int ivoAge = 23;
```

```
int emoAge = 35;  
System.out.print("Gosho is " + ivoAge + " years old.");  
System.out.print("Emo is " + emoAge + " years old.");  
}
```

Резултатът е:

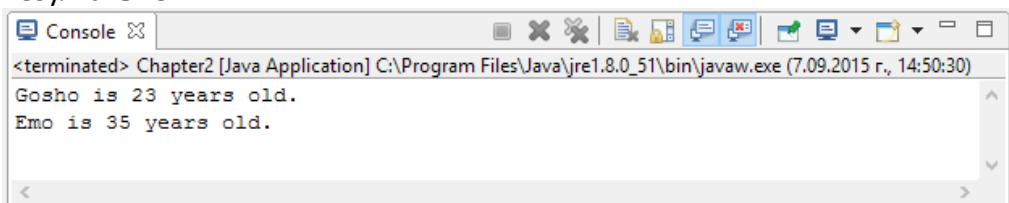


The screenshot shows a console window titled "Console" with the following text:
<terminated> Chapter2 [Java Application] C:\Program Files\Java\jre1.8.0_51\bin\javaw.exe (7.09.2015 г., 14:45:09)
Gosho is 23 years old.Emo is 35 years old.

Нека сега видим същия пример, но чрез използването на **System.out.println()**:

```
public static void main(String[] args) {  
  
    int ivoAge = 23;  
    int emoAge = 35;  
    System.out.println("Gosho is " + ivoAge + " years  
old.");  
    System.out.println("Emo is " + emoAge + " years old.");  
}
```

Резултатът е:

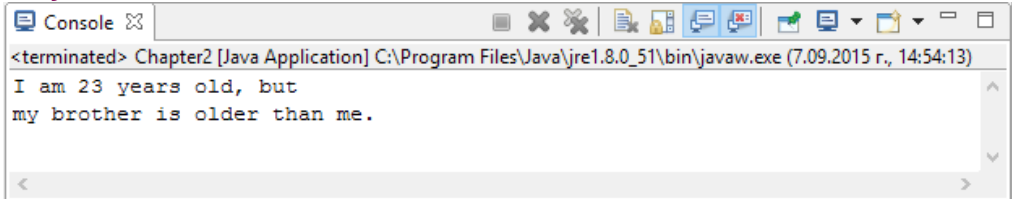


The screenshot shows a console window titled "Console" with the following text:
<terminated> Chapter2 [Java Application] C:\Program Files\Java\jre1.8.0_51\bin\javaw.exe (7.09.2015 г., 14:50:30)
Gosho is 23 years old.
Emo is 35 years old.

Можем да добавяме нов ред чрез изписването на празен **System.out.println()**. Например можем да изпишем едно изречение на два реда, като ние преценим кога искаме да започне новия ред. Това става ето така:

```
public static void main(String[] args) {  
  
    int myAge = 23;  
    System.out.print("I am " + myAge + " years old, but");  
    System.out.println();//нов ред  
    System.out.println("my brother is older than me.");  
}
```

Резултатът е:

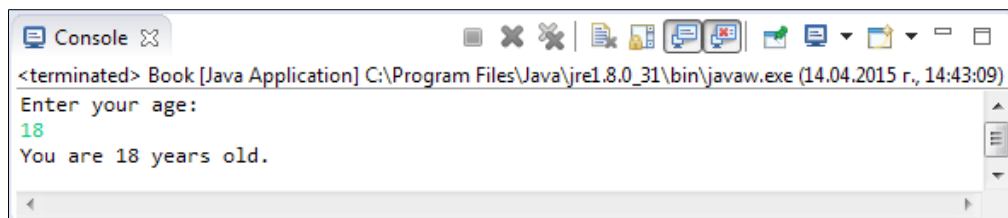


Четене на променливи от конзолата

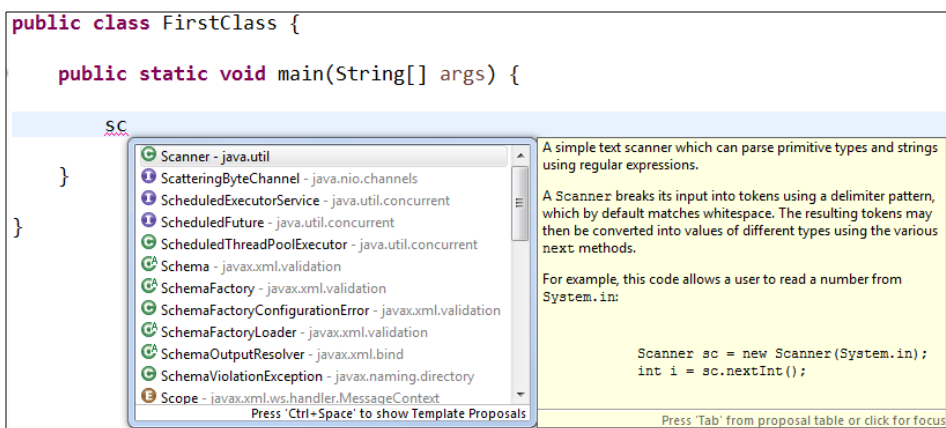
Много често се налага началната стойност на дадена променлива да дойде като входни данни от потребителя, файл, бази данни или друг източник. Четенето на данни в Java от конзолата става посредством използването на конструкция, наречена **Scanner**. Нека да разгледаме един пример как се чете цяло число използвайки Scanner :

```
import java.util.Scanner;  
public class FirstClass {  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter your age:");  
        int age = sc.nextInt();  
        System.out.println("You are " + age + " years  
old.");  
    }  
}
```

Резултатът от програмата е следния:



Първото, което трябва да забележим е **"import"** израза, който включва конструкцията **Scanner** в нашия клас. Scanner-а е конструкцията, която можем да ползваме за да прочетем данни от конзолата. За да имаме достъп до нея, обаче, задължително трябва най-отгоре във файла да напишем „import“ израз. Лесно можете да генерирате този ред като в кода в main напишете "sc" и натиснете Ctrl + Space. От появилото се меню изберете "Scanner – java.util".



Eclipse автоматично ще допълни името на класа, както и ще добави **"import"** декларацията в началото на файла. Тъй като **Scanner** е клас, а класовете не се покриват в тази книга, ще отбележи само това, което е достатъчно да знаем, за да използваме скенера, без да влизаме в подробности. Задаването на начална стойност на променливата **"sc"** става посредством оператора **"new"** и името на класа. Допълнително в скобите инструктираме Scanner **обекта**, който сме създали да чете данни от конзолата ("System.in"). Вече имаме готова конструирана Scanner

променлива, чрез която можем да четем всякакъв тип данни от конзолата. В примера по-горе искаме да прочетем цяло число и затова използваме операцията **sc.nextInt()**. Когато пуснем програмата ще видим, че нищо не се случва. Причината е, че трябва да отидем в конзолата и да въведем някакво цяло число и да натиснем Enter. Въведената стойност от конзолата ще се запише като начална стойност в променливата "age" и в последствие ще бъде изведена заедно със съобщението, което сме поставили в `System.out.println()`. Как можем да четем стойността на променливи от другите примитивни типове? Просто заменяме `nextInt` с `nextDouble`, `nextFloat`, `nextBoolean`, `nextByte` и т.н. в зависимост от типа на променливата, чиято стойност искаме да прочетем. По-долу ще видим пример за програма, която чете стойностите на седем променливи – четири целочислени, две с плаваща запетая и една булева :

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
  
    byte byteVar = sc.nextByte();  
    int intVar = sc.nextInt();  
    short shortVar = sc.nextShort();  
    long longVar = sc.nextLong();  
  
    float floatVar = sc.nextFloat();  
    double doubleVar = sc.nextDouble();  
  
    boolean booleanVar = sc.nextBoolean();  
}
```

Забележете, че създаваме обекта `Scanner` само веднъж в променливата "sc". От там нататък изпълнението на операция от типа `sc.next<тип данни>` води до последващо изчакване на потребителя да въведе данни и прочитането им от конзолата.

Аритметични оператори

Върху променливите от целочислен, с плаваща запетая и знаков тип можем да извършваме следните аритметични операции :

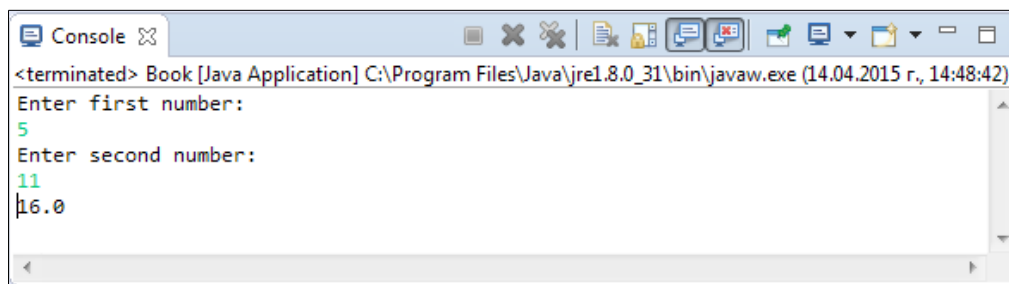
| Оператор | Резултат |
|----------|---|
| + | Събира две променливи |
| - | Изважда две променливи |
| / | Дели променлива на друга |
| * | Умножава две променливи |
| % | Остатък при деление на една променлива на друга |

Нека разгледаме следния пример за програма, която чете две числа с плаваща запетая от конзолата и изкарва сбора им :

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    System.out.println("Enter first number:");  
    double num1 = sc.nextDouble();  
    System.out.println("Enter second number:");  
    double num2 = sc.nextDouble();  
  
    double result = num1 + num2;  
  
    System.out.println(result);  
}
```

Забележете, че на променливата `result` се задава начална стойност (инициализира се) със сбора на двете числа, които сме въвели от клавиатурата.

Резултатът е следния:



```
<terminated> Book [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (14.04.2015 г., 14:48:42)
Enter first number:
5
Enter second number:
11
16.0
```

Оператори за сравнение

В Java можете да сравнявате целите числа, тези с плаваща запетая и знаците използвайки операторите за сравнение. Някои оператори, като тези за равенство и за различие, могат да се прилагат върху променливи от всякакъв тип. Резултатът от прилагането на оператора за сравнение върху две променливи е или **true** или **false**, така че можете да го присвоите на променлива от тип **boolean**. Преди това обаче нека да видим различните оператори за сравнение в Java.

| Оператор | Пример | Описание |
|----------|------------|------------------------------|
| > | $x > y$ | x е по-голямо от y |
| >= | $x \geq y$ | x е по-голямо или равно на y |
| < | $x < y$ | x е по-малко от y |
| <= | $x \leq y$ | x е по-малко или равно на y |
| == | $x == y$ | x е равно на y |
| != | $x != y$ | x е различно от y |



Бъдете внимателни с операторите `==` и `=`. Докато първият оператор сравнява дали две променливи имат еднаква стойност, то втория присвоява стойността от втората променлива на първата.

Ето един пример за програма, която проверява дали сбора на две въведени числа е равен на трето :

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    long num1 = sc.nextLong();  
    long num2 = sc.nextLong();  
    long num3 = sc.nextLong();  
    boolean result = (num1 + num2) == num3;  
    System.out.println(result);  
}
```

Забелязваме, че сборът на променливите **num1** и **num2** е в скоби, защото искаме първо да съберем и след това да проверим, използвайки оператора `==`, за равенство с **num3**. Резултатът от горната програма е **true** или **false** в зависимост от числата, които ще въведем.

Логически оператори

Върху променливите и изразите от булев тип (`boolean`) можем да прилагаме логически оператори. Тези оператори биват :

6. **И** – операторът `&&`
7. **ИЛИ** – операторът `||`
8. **ИЗКЛЮЧАЩО ИЛИ** – операторът `^`
9. **ОТРИЦАНИЕ** – операторът `!`

Как действат тези оператори при различните стойности на булевите променливи? Нека да разгледаме как се държат тези оператори при различните стойности на две променливи А и В :

| A | B | A && B | A B | A ^ B | !A |
|-------|-------|--------|--------|-------|-------|
| true | true | true | true | false | false |
| true | false | false | true | true | |
| false | true | false | true | true | true |
| false | false | false | false | false | |

Ще разгледаме по-подробно приложението на логическите оператори в глава 4 "Условни оператори".

Оператори за присвояване

Освен с оператор равно (=) в Java имаме и други оператори за задаване на нова стойност на променлива. В действителност те са комбинация от оператора равно и някакъв аритметичен оператор, като служат за улеснение и по-бърз запис. Можем да ги видим, заедно с еквивалентите им в таблицата по-долу :

| Оператор | Пример | Еквивалент |
|----------|--------|------------|
| += | x += 2 | x = x + 2 |
| -= | x -= 2 | x = x - 2 |
| *= | x *= 2 | x = x * 2 |
| /= | x /= 2 | x = x / 2 |
| %= | x %= 2 | x = x % 2 |

Оператори за увеличаване и намаляване с единица

Последната група оператори, които ще разгледаме са операторите за увеличаване и намаляване с единица или така наречените пост(пре)инкремент и пост(пре)декремент. Каква е разликата и как се използват можем да видим в таблицата по-долу :

| Оператор | Пример | Описание |
|---------------|--------|---|
| постинкремент | x++ | Връща старата стойност на x, след което увеличава x с 1 |
| преинкремент | ++x | Увеличава x с 1 и след това връща новата стойност |
| постдекремент | x-- | Връща старата стойност на x, след което намалява x с 1 |
| преддекремент | --x | Намалява x с 1 и след това връща новата стойност |

Нека да разгледаме няколко примера за използването на тези оператори:

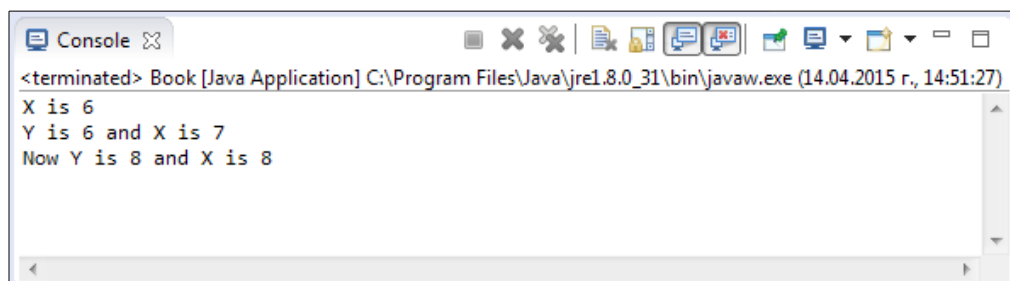
```
public static void main(String[] args) {
    int x = 5;

    x++; //increment x
    System.out.println("X is " + x);

    int y = x++; //take value of x and then increment x
    System.out.println("Y is " + y + " and X is " + x);

    y = ++x; //increment x and then take value of x
    System.out.println("Now Y is " + y +
        " and X is " + x);
}
```

Резултатът е следния:



```
<terminated> Book [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (14.04.2015 г., 14:51:27)
X is 6
Y is 6 and X is 7
Now Y is 8 and X is 8
```

Упражнения

- Да се състави програма, която да извежда квадратите на числата от 1 до 5 на екрана.
- Да се състави програма, която по въведен от конзолата радиус на кръг извежда лицето му.
- Да се състави програма, която по въведени 2 страни на правоъгълник извежда лицето му.
- Да се състави програма, която по въведени 3 страни извежда true или false в зависимост от това дали тези страни могат да образуват триъгълник. Свойството на триъгълника е, че всяка от страните е по-малка от сбора на другите 2. Да се използва оператора логическо И (&&).
- Да се състави програма, която по въведено число извежда true или false в зависимост от това дали числото се дели на 7 без остатък.

Глава 3. Бройни системи. Побитови оператори

В тази глава

Дефиниция на бройна система

Десетична бройна система

Шестнадесетична бройна система

Двоична бройна система

Преобразуване в различни бройни системи

Побитови оператори

Побитови измествания

Дефиниция на бройна система

Бройната система е символичен метод за представяне на числата чрез краен брой символи, наречени цифри. Всяка бройна система има собствени правила за представяне на числата. Бройните системи се разделят спрямо тяхната **основа**. Основата е число, което определя броя на различните цифри, чрез които се записват числата в съответната бройна система.

Бройните системи се разделят на два вида – **позиционни** и **непозиционни**. При позиционните бройни системи разположението на определена цифра в изписването на числото влияе на нейното значение. Непозиционните бройни системи нямат такава зависимост и при тях всяка цифра означава едно и също нещо и има една и съща стойност независимо къде се намира в числото.

Пример за непозиционна бройна система е римската. При нея всяка цифра има точно определена стойност независимо от разположението си в числото.

Числа, представени в определена бройна система, се записват с индекс в долния им край. Индексът е основата на бройната система, чрез която е изписано числото. Пример:

110₍₁₀₎ – сто и десет, изписано в десетична бройна система

110₍₂₎ – шест, изписано в двоична бройна система.

Десетична бройна система (DECIMAL)

Най-разпространената бройна система, която се използва от хората по цял свят, е десетичната бройна система. Основата ѝ е числото 10, което означава, че за записването на всички числа се използват десет цифри (от 0 до 9). Десетичната бройна система е позиционна. Как разбрахме? Нека анализираме две числа: 41 и 31542. При числото 41 стойността на цифрата '1' е едно. При 31542, обаче, стойността (или значението) на цифрата '1' е хиляда. Тоест има значение къде в числото се намира цифрата 1 и позицията на тази цифра (а и на всяка друга) може да повлияе на стойността на цялото число.

Стойността на всяко число може да се изрази като **сума на цифрите, умножени по основата 10 на степен, равна на позицията на цифрата в числото**. Първото число е на нулева позиция, затова го умножаваме по десет на нулева.

Например числото 31542 представлява $30\,000 + 1000 + 500 + 40 + 2$, което всъщност е $3 \cdot 10^4 + 1 \cdot 10^3 + 5 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$.

Двоична бройна система (BINARY)

Двоичната бройна система има за основа цифрата 2. Това означава, че за представянето на всяко число се използват само две цифри – 0 и 1.

Защо ни е нужно да имаме такава бройна система ?

В компютърната техника представянето на числата като комбинации от нули и единици е лесно и удобно. Нулата и единицата в електрониката лесно може да се идентифицира като „има ток“ и „няма ток“ в електрическата верига. Именно така работят съвременните цифрови устройства. Най-малката единица за съхранение на памет в компютърната техника е байт. Един байт се състои от 8 части, наречени битове. Един бит може да има стойност или 0 или 1.

След като знаем това, можем да се досетим, че най-удобният начин за съхранение на данни в паметта е чрез представянето им като поредица от нули и единици. Поради тази причина познанията за двоичната бройна система са от изключително значение.

Пример за число, представено чрез двоичната система е: 110100

В двоичната бройна система не използваме познатите ни от десетичната бройна система цифри – 2,3,4,5,6,7,8 и 9.

Как броим в двоичната бройна система ?

Започваме от нула – 0. Следващото число е едно – 1. А после ?

| Брой | Двоичен вид | Действие |
|------|-------------|-------------|
| нула | 0 | Изписваме 0 |

| Брой | Двоичен вид | Действие |
|------|-------------|---|
| едно | 1 | Продължаваме с 1 |
| две | ??? | Как да постъпим тук ? Нали нямаме цифра 2 ? |

Нека се запитае – как броим в десетичната бройна система ? Изреждаме цифрите от нула до 9, след което започваме отново от нулата, но поставяме цифрата 1 отпред (за да се получи 10 – десет) и пак изреждаме всички останали цифри.

| Десетичен вид | Действие |
|---------------|---|
| 0 | Изписваме 0 |
| ... | Продължаваме с 1, 2, 3, 4, 5, 6, 7, 8, след което ... |
| 9 | Това е последната цифра , която можем да използваме. |
| 10 | Започваме отново от 0, но слагаме 1 в ляво. |

По същия начин постъпваме и при двоичната система - поставяме единица отпред и отново изреждаме цифрите 0 и 1. Аналогично алгоритъма се повтаря и с повече цифри.

| Брой | Двоичен вид | Действие |
|--------|-------------|--|
| нула | 0 | Изписваме 0 |
| едно | 1 | Продължаваме с 1 |
| две | 10 | Отново започваме с 0, но слагаме единица отпред |
| три | 11 | Увеличаваме нулевата позиция с единица |
| четири | 100 | Отново започваме от 0 и добавяме единица вляво. Но в ляво вече има единица, затова и там започва от нула и добавяме единица на следващата позиция вляво. |
| пет | 101 | Увеличаваме нулевата позиция с единица |

| Брой | Двоичен вид | Действие |
|-------|-------------|---|
| шест | 110 | Поставяме единица отпред и започваме отново с нула |
| седем | 111 | Увеличаваме нулевата позиция с единица |
| осем | 1000 | Започваме отново с 0 (за всичките три цифри) и добавяме единица вляво |
| девет | 1001 | Увеличаваме нулевата позиция с единица и т.н. |

Шестнадесетична бройна система (HEXADECIMAL)

Шестнадесетичната бройна система има за основа числото 16. Тази бройна система използва шестнадесет знака за представяне на числата. Първите десет знака са арабските цифри от 0 до 9, а останалите шест – латинските букви A, B, C, D, E и F.

Броенето в шестнадесетична бройна система следва същия принцип като този, който показахме при двоичната. Всички знаци се изреждат, след което вляво се допълва със следващото цифрово означение и подреждането започва отначало.

Пример в следната таблица:

| | | | | | | | | | | | | | | | | | | | | | | | |
|-----|---|-----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DEC | 0 | ... | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| HEX | 0 | ... | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |

Представяне на числа в различна бройна система в Java

В предната глава разгледахме числовите литерали. По подразбиране всички числови литерали в Java се представят в десетичната бройна система. Въпреки това, лесно можем да дефинираме променливи от числов тип и да ги запишем в друга бройна система посредством специален синтаксис.

Двоичен литерал: пред числото записваме **0b**

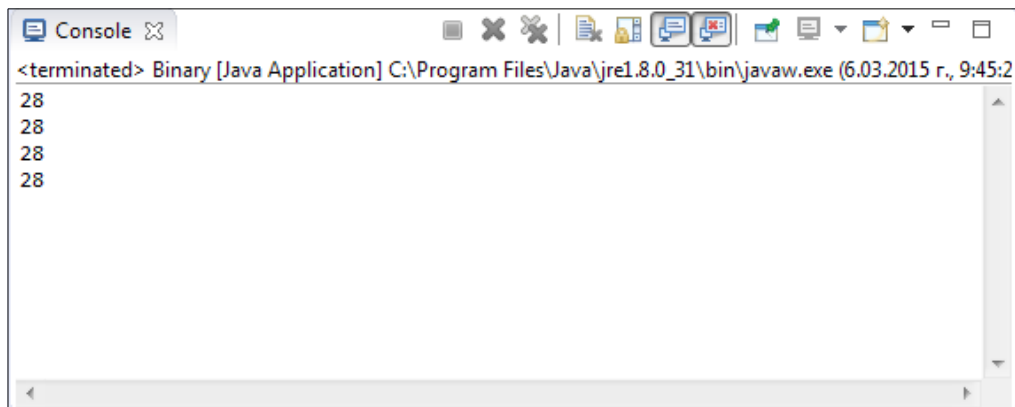
Шестнадесетичен литерал: пред числото записваме **0x**

Осмичен литерал: пред числото записваме **0**

Нека пробваме да дефинираме едно число, записано посредством различни бройни системи:

```
int x = 28; //28 in decimal
int xBin = 0b11100; //28 in binary
int xHex = 0x1C; //28 in hexadecimal
int xOct = 034; //28 in octal decimal
System.out.println(x);
System.out.println(xBin);
System.out.println(xHex);
System.out.println(xOct);
```

В конзолата ще видим изписано числото 28, защото по подразбиране изхода от програмата ни се представя в десетична бройна система:



```
<terminated> Binary [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (6.03.2015 г., 9:45:2
28
28
28
28
```

Преобразуване на числата от една бройна система в друга

Числата могат да се преобразуват от една бройна система в друга. Поради факта, че десетичната бройна система е най-известната и най-

използваната от човека, а двоичната – от машината, от голямо значение е да знаем как да преобразуваме числа от десетична в двоична бройна система и обратното.

Преобразуване от десетична в двоична бройна система

Нека вземем числото $41_{(10)}$ – четиридесет и едно, записано в десетична бройна система. За да може компютърът да запази това число като данна в паметта, то трябва да се преобразува в двоична двойна система. Правилото за преобразуване е следното: Взимаме числото и го делим на основата на системата, към която искаме да го преобразуваме. Записваме остатъка от делението и това ни е една от значещите цифри. След това повтаряме операцията с разделеното вече число и така докато резултата от делението не стане нула. Комбинацията от всички цифри, които сме записали като остатъци от делението, представлява числото в новата бройна система.



Важно е да се отбележи, че остатъците се подреждат отзад напред. Тоест последният получен остатък е най-старшият знак в новото число.

Пример:

$$41 : 2 = 20 \text{ с остатък } 1$$

$$20 : 2 = 10 \text{ с остатък } 0$$

$$10 : 2 = 5 \text{ с остатък } 0$$

$$5 : 2 = 2 \text{ с остатък } 1$$

$$2 : 2 = 1 \text{ с остатък } 0$$

$$1 : 2 = 0 \text{ с остатък } 1$$



Посока на записване на числото.

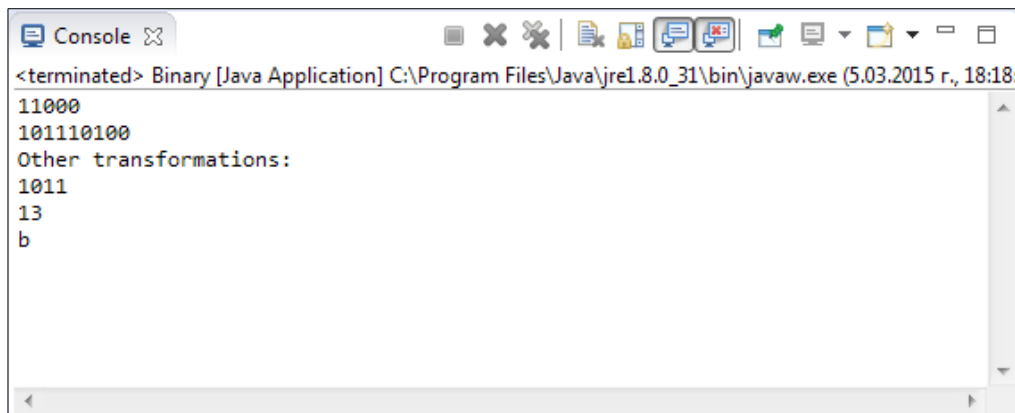
Резултатът от операциите, взет отзад напред, е $101001_{(2)}$. Това е двоичното представяне на числото $41_{(10)}$.

В езика Java има готови вградени инструкции, чрез които можем да преобразуваме числата от десетична в двоична бройна система. Това

става чрез конструкцията **Integer.toBinaryString(x)**, където **x** е числото в десетична бройна система. Друга полезна конструкция е **toString(x,base)**, където **x** е числото в десетична бройна система, а **base** е основата на новата бройна система, към която искаме да преобразуваме числото.

```
public static void main(String[] args) {  
    int number1 = 24;  
    int number2 = 372;  
    System.out.println(Integer.toBinaryString(number1));  
    System.out.println(Integer.toBinaryString(number2));  
    System.out.println("Other transformations:");  
    int number3 = 11;  
    //transform into binary  
    System.out.println(Integer.toString(number3, 2));  
    //transform into octal decimal  
    System.out.println(Integer.toString(number3, 8));  
    //transform into hexadecimal  
    System.out.println(Integer.toString(number3, 16));  
}
```

В конзолата ще можем да видим преобразуваните числа:



```
<terminated> Binary [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (5.03.2015 г., 18:18:  
11000  
101110100  
Other transformations:  
1011  
13  
b
```

Преобразуване от двоична в десетична бройна система

Нека вземем числото $1101101_{(2)}$. За да го преобразуваме в десетична бройна система, е достатъчно да си спомним представянето на двоични числа като сбор от цифрите, умножени по съответните степени на

двойката. Най-младшата цифра (тази най-вдясно) отразява нулевата степен на двойката, втората вляво от нея – първа степен и така до края на числото. След като сме наясно с тази специфика, можем да представим числото така:

$$1101101 = 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

След като решим уравнението вдясно, получаваме

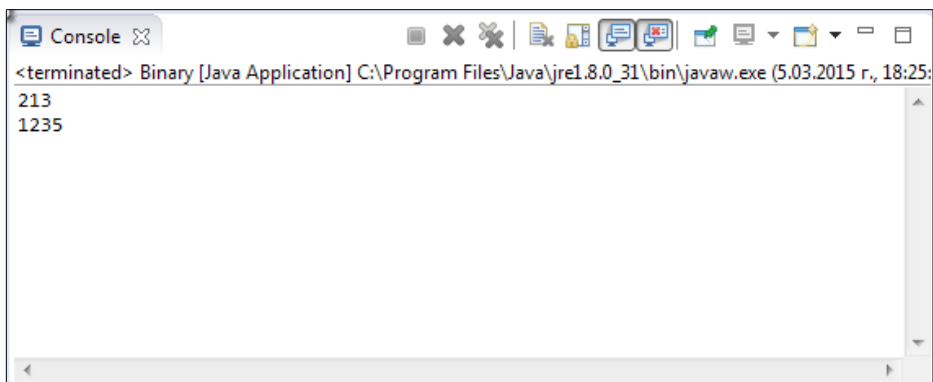
$$1101101 = 64 + 32 + 0 + 8 + 4 + 0 + 1 = 109$$

По този начин открихме, че **1101101**₍₂₎ е равно на **109**₍₁₀₎!

В Java има конструкции, чрез които също можем да преобразуваме двоични числа в десетична бройна система. Това става чрез конструкцията **Integer.parseInt(number, base)**, където **number** е числото, което искаме да обърнем към десетична бройна система, а **base** е основата на оригиналната бройна система, чрез която е представено числото. Забележете, че е нужно да подадем числото в кавички, за да работи тази конструкция.

```
public static void main(String[] args) {  
    //from binary to decimal  
    System.out.println(Integer.parseInt("11010101", 2));  
    //hexadecimal to decimal  
    System.out.println(Integer.parseInt("4D3", 16));  
}
```

В конзолата ще можем да видим преобразуваните числа:



Операции с двоични числа. Побитови оператори

В предната глава разгледахме стандартните аритметични операции с примитивните типове данни. Понеже данните се пазят в паметта под формата на единици и нули, вътрешно всяка аритметична операция се свежда до операция на двоичните записи на данните. Например ако искаме да направим сбор на числата 3 и 5, резултата е 8. Но вътрешно този сбор представлява набор от операции с двоично представените числа 3 и 5 и преобразуване на резултата обратно до десетичното 8, за да се появи на екрана.

Двоичните числа се представят в паметта като поредица от битове. Всеки бит съхранява единица или нула. Понеже операциите с двоични числа представляват операции върху всеки бит, ги наричаме побитови операции.

Оператор за побитово отрицание (NOT)

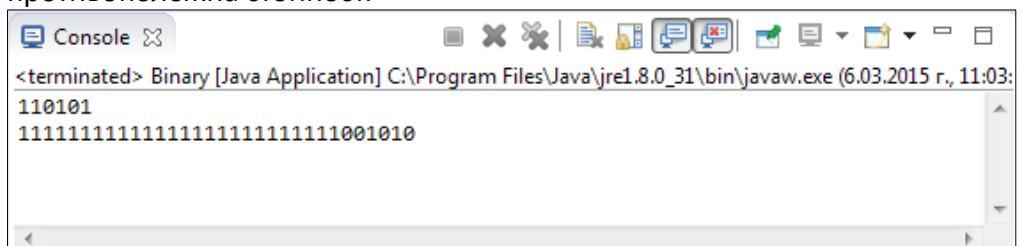
Побитовото отрицание се извършва на едно число, като всеки бит от това число се инвертира (обръща). Ако бита е бил единица, става нула и ако е бил нула – става единица.

Оператора в Java за отрицание е '~' (тилда).

Нека разгледаме двоичното число **110101**₍₂₎. Чрез оператора тилда можем да го инвертираме и да покажем резултата в конзолата:

```
int number = 0b110101;
int invertedNumber = ~number;
System.out.println(Integer.toBinaryString(number));
System.out.println(Integer.toBinaryString(invertedNumber));
```

Като резултат получаваме двоично число, при което всеки от битовете е с противоположна стойност.



Забележете, че инвертираното число има доста единици пред себе си. Откъде се взеха ? Ако си спомняте от предната глава, примитивният тип **int** е 32-битов. Това значи, че за съхранението в паметта на едно число от тип **int** се използват 32 бита, което означава комбинация от 32 нули и единици. Нашето число 110101 де факто е предшествано от 26 нули, но конзолата не ги изписва, защото те просто не са значещи. Числото едно от тип **int** в двоичен вид всъщност се представя така: 000000000000000000000000000001. Тридесет и едната нули отпред просто няма смисъл да се визуализират, защото не се отразяват на стойността на числото.

Когато инвертираме число, обаче, ние инвертираме всеки от битовете, включително и нулите, които конзолата не показва. Именно затова имаме много единици в изписването на променливата **invertedNumber**.

Оператор за побитово И (AND)

Побитово И се извършва между две числа. Действието се отразява на двойката битове от всяко число в съответната позиция. Получения резултат е нов бит, като правилото е следното:

Ако и двата бита са единица, то получения бит е единица. В останалите случаи получения бит е нула.

Операторът за побитово И в Java е **'&'** (амперсанд).

Пример – искаме да извършим побитово И на числата 110101₍₂₎ и 111000₍₂₎.

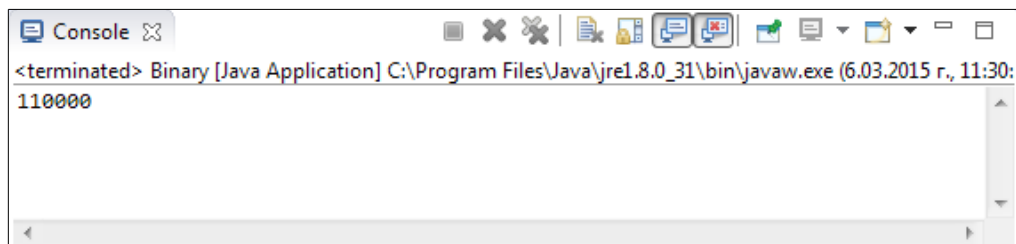
Извършваме операцията на всяка двойка битове и конструираме ново число, получено от отделните резултати:

| | | | | | | |
|-----------------|---|---|---|---|---|---|
| число1 | 1 | 1 | 0 | 1 | 0 | 1 |
| число2 | 1 | 1 | 1 | 0 | 0 | 0 |
| число1 & число2 | 1 | 1 | 0 | 0 | 0 | 0 |

Гореописаната операция, представена в Java изглежда така:

```
int x = 0b110101;
int y = 0b111000;
int z = x & y;
System.out.println(Integer.toBinaryString(z));
```

В конзолата можем да видим резултата **110000₍₂₎**



Оператор за побитово ИЛИ (OR)

Побитово ИЛИ също се извършва между две числа. Действието се отразява на двойката битове от всяко число в съответната позиция. Получения резултат е нов бит, като правилото е следното:

Ако поне един от двата бита е единица, то получения бит е единица. В останалите случаи получения бит е нула.

Операторът за побитово ИЛИ в Java е '|' (една права черта).

Нека вземем същите две числа от предишния пример - 110101₍₂₎ и 111000₍₂₎ и извършим побитово ИЛИ.

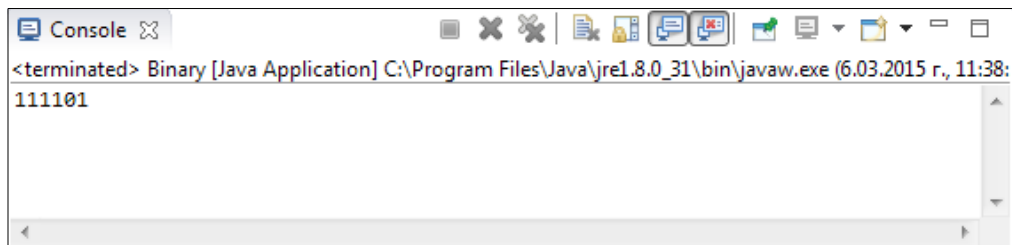
Извършваме операцията на всяка двойка битове и конструираме ново число, получено от отделните резултати:

| | | | | | | |
|-----------------|---|---|---|---|---|---|
| число1 | 1 | 1 | 0 | 1 | 0 | 1 |
| число2 | 1 | 1 | 1 | 0 | 0 | 0 |
| число1 число2 | 1 | 1 | 1 | 1 | 0 | 1 |

Гореописаната операция, представена в Java изглежда така:

```
int x = 0b110101;
int y = 0b111000;
int z = x | y;
System.out.println(Integer.toBinaryString(z));
```

В конзолата можем да видим резултата **111101**₍₂₎



Оператор за изключващо ИЛИ (XOR)

Побитово изключващо ИЛИ също се извършва между две числа. Действието се отразява на двойката битовете от всяко число в съответната позиция. Получения резултат е нов бит, като правилото е следното:

Ако двата бита са с различни стойности, то получения бит е единица. В останалите случаи получения бит е нула.

Операторът за побитово изключващо ИЛИ в Java е '^' (колибка).

Нека вземем същите две числа от предишния пример - 110101₍₂₎ и 111000₍₂₎ и извършим побитово изключващо ИЛИ.

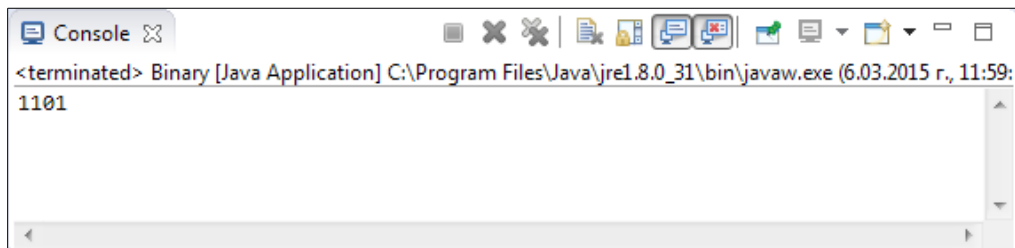
Извършваме операцията на всяка двойка битовете и конструираме ново число, получено от отделните резултати:

| | | | | | | |
|-----------------|---|---|---|---|---|---|
| число1 | 1 | 1 | 0 | 1 | 0 | 1 |
| число2 | 1 | 1 | 1 | 0 | 0 | 0 |
| число1 ^ число2 | 0 | 0 | 1 | 1 | 0 | 1 |

Гореописаната операция, представена в Java изглежда така:

```
int x = 0b110101;  
int y = 0b111000;  
int z = x ^ y;  
System.out.println(Integer.toBinaryString(z));
```

В конзолата можем да видим резултата **111101**₍₂₎



Побитови измествания

Побитовите измествания са операции, при които стойностите на всеки бит от числото се изместват в съответно наляво или надясно.

В Java операторите за побитово изместване са съответно '»' за изместване надясно и '«' за изместване наляво.

Изместването е операция, която има нужда от два операнда – числото, което ще измества и броя позиции, с които ще се измества.

Важно е да се отбележи, че при изместването наляво, на мястото на първия бит се слага нула. Най-левия бит изчезва от числото.

Пример: 110101 « 1 = 1101010

В примера най-левия бит не изчезва, защото приемаме, че работим с int числа, а те имат 32 бита. Следователно пред гореописаните числа има нули, но не ги визуализираме.

Ако изместването е надясно, то първия бит изчезва от числото, независимо каква стойност е имал.

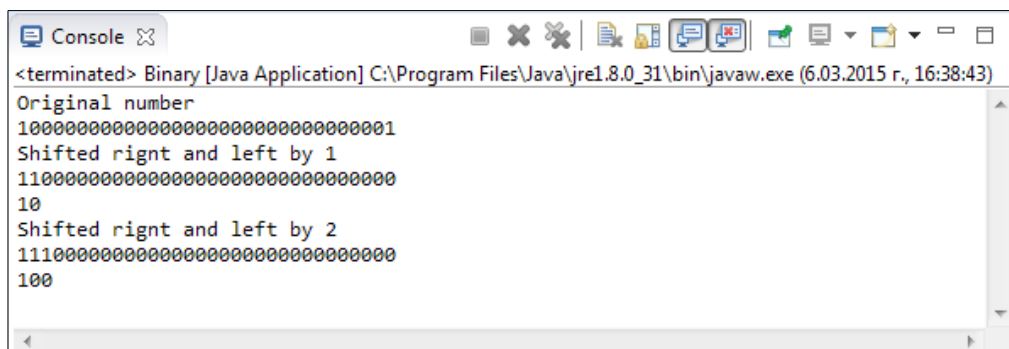
Пример: 110101 » 1 = 11010

Ако, обаче, най-старшият бит е единица, след изместването му надясно числото се допълва отново с единица.

Нека приложим операторите за преместване в Java:

```
int x = 0b10000000000000000000000000000001;  
int xOneRight = x >> 1;  
int xOneLeft = x << 1;  
System.out.println("Original number");  
System.out.println(Integer.toBinaryString(x));  
System.out.println("Shifted right and left by 1");  
System.out.println(Integer.toBinaryString(xOneRight));  
System.out.println(Integer.toBinaryString(xOneLeft));  
int xTwoRight = x >> 2;  
int xTwoLeft = x << 2;  
System.out.println("Shifted right and left by 2");  
System.out.println(Integer.toBinaryString(xTwoRight));  
System.out.println(Integer.toBinaryString(xTwoLeft));
```

В конзолата ще видим следния резултат:



```
<terminated> Binary [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (6.03.2015 г., 16:38:43)  
Original number  
10000000000000000000000000000001  
Shifted right and left by 1  
11000000000000000000000000000000  
10  
Shifted right and left by 2  
11100000000000000000000000000000  
100
```

Упражнения

1. Чрез побитови операции изведете от числото 3, числото 7.
2. Чрез побитови операции изведете от числото 231, числото 772
3. Напишете програма, която приема за входен параметър число и извежда в конзолата колко бита са единица в това число.
4. Напишете програма, която да проверява дали петия бит на дадено число е нула или едно.

5. Напишете програма, която да проверява колко бита от две входни числа са с еднаква стойност.

Глава 4. Условни оператори

В тази глава

Условен оператор if

Оператор за множество условия switch-case

Оператор за сравнение ?:

Комбиниране на няколко условия използвайки логически оператори

Условен оператор if

Много често ни се налага да изпълним определен блок от код само в някои случаи. Например : ако потребителят е загубил играта, да му покажем съобщение да започне отначало; ако даден файл бъде свален, да му го отворим и т.н. За тази цел в Java, както и в повечето популярни езици за програмиране имаме условен оператор **if**, който на базата на някакво зададено условие изпълнява или не даден блок от код. В най-простата си форма **if** конструкцията изглежда по следния начин :

```
if (условие) {  
    //блок от код, който да се изпълни  
    //когато това условие е вярно  
}
```

Условието, което поставяме в скобите е израз, чийто резултат е от тип **boolean** или с други думи има стойност истина (true) или лъжа (false).

Пример за такива изрази са :

- Логическо условие – $a > 10$, $b < 5$, $c \neq 4$
- Булева променлива – `if (a) {...}`, където `a` е променлива от тип `boolean`
- Обединяване на няколко условия, използвайки логически оператори, които ще разгледаме по-нататък в тази глава.

Като демонстрация на оператора **if** ще разгледаме кратка програма, която чете число от клавиатура и извежда дали числото е положително :

```
import java.util.Scanner;
public class IfDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int number = sc.nextInt();
        if (number > 0) {
            System.out.println("The number is positive");
        }
    }
}
```

Никога не поставяйте ; след скобите с условието на оператора if. Това ще доведе до това блока от код да се изпълнява винаги независимо какво е условието.

Следният код винаги ще изкарва, че сме въвели положително число, независимо какво въвеждаме :



```
Scanner sc = new Scanner(System.in);
int number = sc.nextInt();
if (number > 0); { //не слагайте ; след условието в if
    //това винаги ще излиза
    System.out.println("The number is positive");
}
```

Оператор if-else

Като допълнение към оператора if можем да добавим **else**. След **else** пишем блок от код, който се изпълнява, когато условието не е вярно. **if-else** конструкцията изглежда по следния начин :

```
if (условие) {  
    //блок от код, който да се изпълни  
    //когато резултата от условието е истина  
} else {  
    //блок от код, който да се изпълни  
    //когато резултата от условието е лъжа  
}
```

Ако се върнем на примера от по-горе, можем да изведем дали въведеното число е положително или не, защото в противен случай при въведено отрицателно число няма да получим никакъв резултат :

```
import java.util.Scanner;  
  
public class IfDemo {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        int number = sc.nextInt();  
  
        if (number > 0) {  
            System.out.println("Числото е положително");  
        } else {  
            System.out.println("Числото е 0 или  
отрицателно");  
        }  
    }  
}
```



Винаги подравнявайте кода в блока на `if` или `else` с една табулация (с клавиша `tab`) навътре, почвайки на нов ред. Това прави кода по-лесно четим и разбираем. Компилятора няма да генерира грешка в противен случай, но за вас и вашите колеги това няма да е много приятен за работа код. Например:

```
if (a>b) {  
    System.out.println("a е по-голямо"); m = a;  
}  
else  
{  
    System.out.println("b е по-голямо");  
    m = b;}
```

Вложени if оператори

Нямаме никакви ограничения за това колко и какви оператори можем да използваме в блоковете от кода за `if` и `else`. От това следва, че можем вътре в тялото на даден `if` или `else` да напишем нова условна конструкция `if` или `if-else`. Обикновено наричаме това "вложени `if` оператори". Ако се върнем на примера от по-горе, можем да преправим кода така, че да изведем дали това, което сме въвели, е положително число, отрицателно число или нула.

```

Scanner sc = new Scanner(System.in);
int number = sc.nextInt();
if (number > 0) {
    System.out.println("Числото е положително");
}
else {
    if (number == 0) {
        System.out.println("Числото е равно на 0");
    } else {
        System.out.println("Числото е отрицателно");
    }
}

```

Не е добра практика да влагаме повече от два if оператора, защото кода става неясен, трудно четим с логика трудна за проследяване.

Оператор switch-case

За да илюстрираме нуждата от следващия вид условен оператор, ще разгледаме следната примерна задача – по даден ден от седмицата, число между 1 и 7, да се изведе името на съответния ден. Например при въведено 1 - Понеделник, 2 – Вторник, 3 - Сряда и т.н. Можем да реализираме това, използвайки седем if конструкции и съответно за всеки ден от седмицата да изкарваме името на този ден, а при липса на съвпадение, да изведем, че няма ден с такъв номер :

```

import java.util.Scanner;

public class SwitchDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int dayOfWeek = sc.nextInt();
        if (dayOfWeek == 1) {
            System.out.println("Понеделник");
        } else
        if (dayOfWeek == 2) {

```

```

        System.out.println("Вторник");
    } else
    if (dayOfWeek == 3) {
        System.out.println("Сряда");
    } else
    if (dayOfWeek == 4) {
        System.out.println("Четвъртък");
    }
    if (dayOfWeek == 5) {
        System.out.println("Петък");
    } else
    if (dayOfWeek == 6) {
        System.out.println("Събота");
    } else
    if (dayOfWeek == 7) {
        System.out.println("Неделя");
    } else {
        System.out.println("Няма такъв ден");
    }
}
}

```

Недостатък на горното решение е многото повторение на код, както и множеството излишен код, който се пише. Можем да съкратим горното решение и да го направим по-ясно и четливо използвайки оператора switch-case. Операторът има следния вид :

```

switch (променлива) {
    case конкретна_стойност1:
        //код,който се изпълнява, когато променливата е равна
на конкретна_стойност1
        break;
    case конкретна_стойност2:
        //код,който се изпълнява, когато променливата е
равна на конкретна_стойност2
        break;
    //....
}

```

```
    default:
        //код,който се изпълнява, когато променливата не е
        равна на никоя от разглежданите стойности
        break;
}
```

Операторът **switch** приема дадена променлива и започва последователно да сравнява стойността ѝ със стойностите, които сме задали като случаи (**case**). При съвпадение, съответния блок от код се изпълнява, а иначе се продължава с проверка на другите стойности, които сме указали. Ако никоя стойност не съвпадне със стойността на променливата в **switch**, то се изпълнява блока от код в **default** секцията. Ако намерим съвпадение в някой **case**, то освен този код се изпълнява и всичко надолу до края на оператора. Можем да преработим примера от началото на тази секция по следния начин използвайки **switch-case** оператора :

```
switch (dayOfWeek) {
case 1:
    System.out.println("Понеделник");
    break;
case 2:
    System.out.println("Вторник");
    break;
case 3:
    System.out.println("Сряда");
    break;
case 4:
    System.out.println("Четвъртък");
    break;
case 5:
    System.out.println("Петък");
    break;
case 6:
    System.out.println("Събота");
}
```

```

        break;
    case 7:
        System.out.println("Неделя");
        break;
    default:
        System.out.println("Няма такъв ден от седмицата");
        break;
}

```

Операторът **break**, се използва, за да прекъснем изпълнението на оператора **switch**, след като е намерена съответната стойност в **case**. Ако **break** липсва, то кода ще продължи да се изпълнява от намерената стойност надолу до края на **switch** оператора, включително и кода указан в **default** частта. Това може да доведе до неочаквани резултати, поради което не трябва да забравяме на указваме **break** в края на всеки случай, който искаме да се изпълни. Като добра практика е хубаво да се слага **break** дори и след default секцията. Това поведение на оператора **switch-case** може да бъде полезно в случаите, когато искаме един и същ код да се изпълнява за няколко различни случая. Като пример ще дадем следната програма, която изкарва за въведен месец броя дни в този месец ако приемем, че годината е високосна.

```

public class SwitchDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int month = sc.nextInt();
        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                System.out.println(31);
                break;
            case 2:

```



```

        System.out.println(29);
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println(30);
        break;
    default:
        System.out.println("Няма такъв номер на
месец");
        break;
    }
}
}

```

Как работи горния пример? Когато въведем в конзолата номер на месец, например 3, **switch-case** оператора започва да търси дали има случай (**case**), който да отговаря на тази стойност. Тъй като имаме такъв случай, кода започва да се изпълнява от тази стойност (case 3:) надолу. Нямаме **break** и кода продължава като се изпълняват случаите за стойности 3, 5, 7, 8, 10 и 12. В случая за 12 вече извеждаме, че имаме 31 дни в този месец и прекратяваме изпълнението на **switch** оператора с **break**. По този начин ние логически групираме месеците с 31 дни, като им задаваме един и същ код, който да се изпълнява при тях. Кодът работи по аналогичен начин месеците с 30 дни, а отделно сме поставили случая за месец февруари, тъй като той не попада нито в месеците 30, нито в тези с 31 дни.

Оператор за сравнение ?:

Понякога искаме да проверим дадено условие и в зависимост от него да присвоим една или друга стойност на дадена променлива. Ще илюстрираме това със следния пример :

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
int c;
if (a>b) {
    c = a;
}
else {
    c = b;
}
```

Променливата *c* взема стойността на по-голямото от двете въведени стойности в променливите *a* и *b*. Използвайки тернарния оператор `?:` можем да съкратим значително горния код. Операторът има следният вид :

**променлива = (условие) ? стойност при истина : стойност при
лъжа;**

Стойността, която се присвоява на променливата е равна на стойността след знака "`?`", ако условието е истина, или стойността след знака "`:`" ако условието е лъжа. Прилагайки това към примера от по-горе можем да разпишем кода по следния начин :

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
int c = (a>b) ? a : b;
```

Не е добра практика да влагаме няколко оператора `?:`, тъй като кода става неясен и трудно четим. Обикновено го използваме за кратки и ясни изрази в случаите, в които трябва да вземем една или друга стойност на база на някакво просто условие.

Комбиниране на няколко условия, използвайки логически оператори

Можем да комбинираме няколко условия използвайки операторите **&&** и **||**, споменати по-рано в глава 2-ра. Първият оператор ще даде стойност истина и блока от код в **if** ще бъде изпълнен ако и двете условия бъдат верни. За вторият оператор, за да получим истина е достатъчно само едно от двете условия да бъде вярно. Нека да разгледаме следния пример, в който проверяваме дали въведено число представлява валидни години на човек :

```
Scanner sc = new Scanner(System.in);
int age = sc.nextInt();
if (age > 0 && age <= 100) {
    System.out.println("Валидни години за човек");
}
else {
    System.out.println("Невалидни години за човек");
}
```

В примера от по-горе първия блок от код ще бъде изпълнен, когато въведената стойност в променливата за години бъде едновременно по-голяма от 0 и по-малка или равна на 100. Т.е. оператора **&&** (логическо И) изисква едновременно и двете условия да бъдат истина. От друга страна оператора **||** (логическо ИЛИ) ще влезе в тялото на **if** оператора, ако 1 или повече от условията са верни. Можем да променим примера от по-горе, за да проверим дали даден човек е в трудоспособна възраст или не.

```

Scanner sc = new Scanner(System.in);
int age = sc.nextInt();
if (age < 18 || age > 65) {
    System.out.println("Човека е в неработоспособна възраст.");
}
else {
    System.out.println("Човека е в работоспособна възраст.");
}

```

Последния логически оператор, който ще демонстрираме е оператора за логическо отрицание (! - логическо НЕ). Много често искаме нещо да се случи, когато дадено условие не е изпълнено. Можем да "обърнем" условието, като поставим знака ! пред него и го заградим в още едни кръгли скоби. Следва пример, в който ще проверим дали стойността на дадена променлива от знаков тип, не е цифра.

```

char symbol = 'W'; //примерна стойност, която сме въвели

if ( !(symbol >= '0' && symbol <= '9') ) {
    System.out.println("Въведенията символ не е цифра.");
}

```

Първо трябва да отбележим, че тъй като променливите от тип **char** всъщност представляват номера на символи в съответната кодова таблица, то е напълно валидно да сравняваме дадена променлива с операторите <=, <, >, >= заедно с някакъв символ. В случая искаме да проверим дали даден символ не е цифра. Тъй като знаците в кодовата таблица са наредени последователно, то символа ще е цифра, ако е между кодовете на знаците за 0 и 9. Ние искаме да изведем съобщение, за неправилно въведен символ в обратния случай, така че пред условието поставяме знака за логическо отрицание ! и така кодът в блока на условния оператор **if** се изпълнява точно когато това условие не е изпълнено.

Упражнения

- Да се състави програма, която прочита 3 числа и извежда кое от тях е най-голямо.
- Да се състави програма, която прочита 3 числа и извежда дали второто е между първото и третото.
- Да се състави програма, която въвежда 2 променливи от клавиатурата сума пари-число с плаваща запетая и дали съм здрав – булев тип. Нека програмата изкарва решения на базата на тези данни по следния начин:
 - ако съм болен - няма да излизам
 - ако имам пари - ще си купя лекарства
 - ако нямам пари – ще стоя вкъщи и ще пия чай
 - ако съм здрав
 - ако имам над 10 лв. - ще отида на кино с приятели
 - ако имам по-малко от 10 лв. - ще отида на кафе.

Полученото решение покажете като съобщение в конзолата.

Глава 5. Цикли

В тази глава

Дефиниция на цикъл

Цикъл „while“

Цикъл „do-while“

Цикъл „for“

Вложени цикли

Ключови думи „break“ и „continue“

Дефиниция за цикъл

Нека вземем предвид следния проблем: Да се изведат на екрана числата от едно до три в нарастващ ред. Това е лесно, нали ? Просто изписваме на всеки отделен ред числата от едно до три чрез конструкцията **System.out.println(...)**.

Как бихме променили, обаче, кода си, ако имаме за задача да изпишем числата от едно до три хиляди ? Три хиляди конструкции **System.out.println(...)** биха решили задачата, нали ? А удобно ли е това решение ?

Нека вземем друг пример – да се изведат всички числа от едно до N, където N е число, въведено от потребителя. В такъв случай дори не знаем колко конструкции за принтиране в конзолата ни трябва.

За решение на тези и други подобни проблеми, в програмирането са въведени специфични конструкции, наречени цикли.

Цикълът е конструкция, която ни позволява да изпълним дадено парче код много на брой пъти.

Можем да кажем, че цикълът се състои от следните компоненти:

1. **тяло на цикъла (body)** – това е кодът, който имаме нужда да повторим определен брой пъти;
2. **условие на цикъла (condition)** – това е булева променлива или израз с булева стойност, според която се определя дали тялото на цикъла да се изпълни отново;



Едно изпълнение на тялото на цикъл наричаме итерация. Тоест ако един цикъл изпълнява определено парче код 10 пъти, то тогава казваме, че цикълът има 10 итерации.

Има два варианта за изпълнение на цикъл:

- фиксиран брой пъти (фиксиран брой повтарения на тялото на цикъла);
- докато някакво условие е в сила.



Цикъл, при който условието винаги е в сила, се нарича безкраен цикъл. При изпълнение на безкраен цикъл програмата никога няма да спре изпълнението си.

Нека вземем за пример един от горепосочените проблеми – да се изпишат на екрана числата от 1 до 3000.

Как би изглеждал този код чрез стандартните конструкции за принтиране на текст в конзолата ?

```
System.out.println(1);
System.out.println(2);
System.out.println(3);
System.out.println(4);
System.out.println(5);
System.out.println(6);
//..още много такива редове с всички числа
System.out.println(2999);
System.out.println(3000);
```

Гореописаният код може да се сведе до това:

```
int x = 1;
System.out.println(x);
x++;
System.out.println(x);
x++;
System.out.println(x);
x++;
```



```
//... и така 3000 пъти
```

Забележете, че тук се повтаря едно и също нещо 3000 пъти.

Циклите са замислени така, че едно парче код, което имаме нужда да се повтаря много на брой пъти, да се изпише само веднъж. По този начин кода става по-четим, по-добре структуриран и по-лесно поддържан.

Защо ? Представете си, че в гореописаното решение някой идва при вас и ви казва да промените програмата така, че числата да се извеждат през едно. Тогава на 3000 места в програмата трябва вместо **x++** да напишете **x+=2**. Това ще ви отнеме известно време и много нерви. Именно такива ситуации трябва да се избягват и затова циклите са изключително важен инструмент в арсенала на всеки програмист.

Съществуват три различни конструкции, чрез които може да се реализира цикъл в Java.

Цикъл „while“

Споменахме, че чрез циклите можем да изпълняваме едно парче код **докато** е изпълнено някакво условие. Тук наблягаме на думата „**докато**“, защото точно така звучи и най-простата конструкция за цикъл – **while**.

Цикълът „**while**“ представлява конструкция, която изглежда по следният начин:

```
while( условие ) {  
    //парче код;  
    //парче код;  
}
```

Конструкцията се състои от следните три неща:

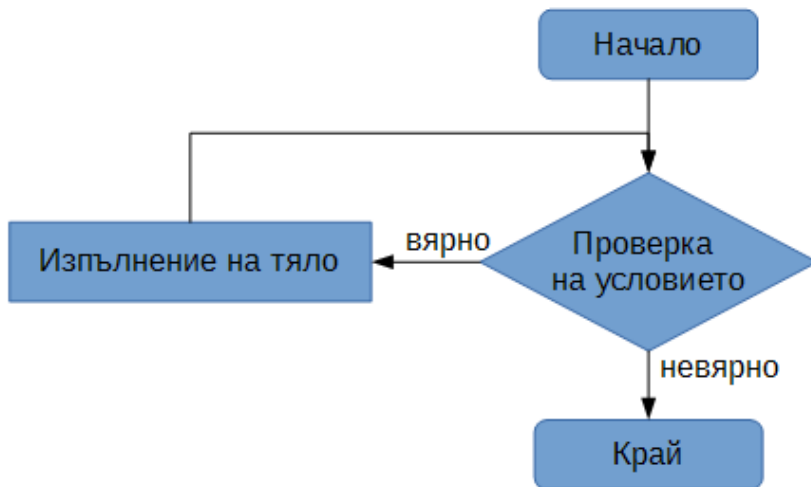
- Ключова дума **while** – така компилатора разбира, че следва конструкция на цикъл.
- Условие на цикъла (**condition**) – това е булев израз или стойност на булева променлива.
- Тяло на цикъла (**body**) – това е парчето код, което искаме да повтаряме.

While цикълът работи по следния начин: **докато** (**while**) е в сила някакво условие (**condition**), изпълнявай следният код (**body**).

Можем да визуализираме какво се случва по следния начин:

```
докато( това е вярно ) {  
    изпълнявай това;  
}
```

Следната блок схема визуализира работата на „while“ цикъла:



Нека вземем за пример нашият проблем с изписването на числата от едно до 3000. Задачата можем да реализираме по следният начин:

```
int x = 1;  
while( x <=3000 ) {  
    System.out.println(x);  
    x++;  
}
```

Как работи гореописаният код ? Инициализираме една променлива `x` със стойност 1. След това казваме докато `x` е по-малко или равно на 3000, да се изпълнява следното парче код: изпиши `x` на екрана и увеличи `x` с единица.

След като се изпълни тялото на цикъла, се проверява отново условието, зададено след ключовата дума `while`. Ако условието все още е в сила, тялото на цикъла се изпълнява отново и т.н. докато дойде момент, в който условието на цикъла вече няма да е вярно. В нашият случай след като `x` стане 3000, то променливата ще се изпише на екрана и ще се

инкрементира. В този момент `x` ще стане 3001 и проверката на условието ще установи, че то вече не е в сила. Тогава тялото на цикъла няма да се изпълни и програмата ще продължи напред.



При цикълът „While“ тялото се изпълнява след проверка на условието. Тоест ако при стартиране на цикъла условието не е в сила, тялото на цикъла няма да се изпълни нито веднъж.

Ако искаме да променим нашата програма и тя да принтира числата от 1 до 2 000 000 000, то тогава просто променяме условието на цикъла – `x` да е по-малко или равно на два милиарда.

```
int x = 1;
while( x <= 2_000_000_000) {
    System.out.println(x);
    x++;
}
```

Ако искаме да отпечатваме всички числа от 1 до 3000, които се делят на 5, тогава ще променим единствено тялото на цикъла по следния начин:

```
int x = 1;
while( x <= 3000) { //докато x не стане 3000
    if(x%5==0) { //принтираме x само ако се дели на 5
        System.out.println(x);
    }
    x++; //винаги увеличаваме x с едно
}
```

Забележете с колко малко на брой промени можем да изменим функционалността на нашата програма, без да има повтаряне на код.

Нека помислим и над втората задача - да се изведат всички числа от едно до `N`, където `N` е число, въведено от потребителя. За целта вместо да инициализираме `x` с 1, ще го инициализираме със стойност, въведена от потребителя.

```
Scanner scan = new Scanner(System.in);
int number = scan.nextInt();
int x = 1;
```

```
while( x <= number) {  
    System.out.println(x);  
    x++;  
}
```

Как би изглеждал безкраен **while** цикъл ? Има два начина:

1. Подаваме **true** като условие на цикъла

```
while( true ) { //условието винаги е изпълнено  
    System.out.println("Hello!"); //безкраен печат  
}
```

2. Подаваме условие, което винаги ще бъде в сила. Това можем да го постигнем по много начини

- ♦ ако условието зависи от променлива, но в тялото на цикъла ние не променяме стойността на тази променлива:

```
int x = 1;  
while( x <= 5) { //x е винаги 1  
    System.out.println(x); //безкраен печат  
}
```

- ♦ когато дори да променяме променливата, това няма да се отрази на условието:

```
int x = 1;  
while( x < 5) { //x е винаги по-малко от 5  
    System.out.println(x); //безкраен печат  
    x--; //намаляваме x с едно  
}
```

Цикъл „do-while“

Конструкцията „do-while“ е аналогична на „while“ и изглежда по следния начин:

```
do {  
    //парче код;  
    //парче код;  
}  
while( условие );
```

Конструкцията се състои от следните четири неща:

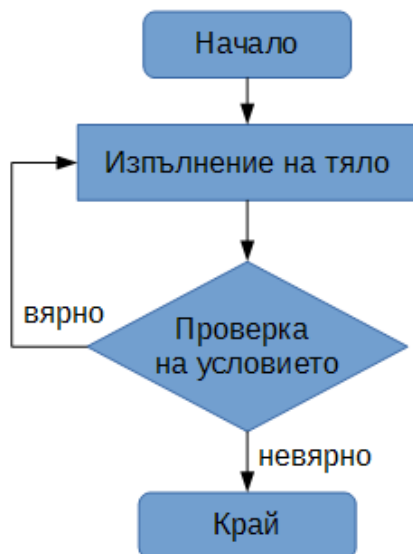
1. ключовата дума **do** – така компилатора разбира, че следва конструкция на цикъл
2. тялото на цикъла (**body**) – това е кода, който се изпълнява веднъж и се повтаря, ако е изпълнено условието на цикъла
3. ключовата дума **while** – непосредствено след тялото на цикъла
4. условието (**condition**), според което тялото на цикъла трябва да се повтаря.

Do-while цикълът работи по следния начин: изпълни (точка 1) следният код (точка 2). Докато (точка 3) е в сила някакво условие (точка 4), повтори вече изпълнения код (точка 1) отново.

```
изпълни {  
    това парче код;  
}  
докато( това е вярно )
```

Съществената разлика между **do-while** и **while** конструкцията е, че при **do-while** тялото на цикъла се изпълнява **ПРЕДИ** да се провери условието. Ако условието на цикъла е изпълнено, тогава тялото се повтаря и т.н. докато условието на цикъла не стане **false**. По този начин do-while конструкцията ни гарантира изпълнение на тялото на цикъла най-малко веднъж.

Следната блок схема визуализира работата на „do-while“ цикъла:



Основната характеристика на цикълът „do-while“ е, че дори при неизпълнено условие, тялото на цикъла се изпълнява поне веднъж.

Удачен пример за използване на do-while вместо while конструкция е проблема с въвеждането и валидирането на данни от потребител. Ако въведените данни не са валидни, ние трябва отново да изискаме от потребителя да въведе нови данни. Все пак, потребителят трябва да въведе данни преди условието за валидност да е проверено.

Решението изглежда така:

```

Scanner scan = new Scanner(System.in);
int input = 0;
do {
    System.out.println("Въведете число между 1 и 10");
    input = scan.nextInt();
}
while(input < 1 || input > 10);
  
```

Нека разгледаме подробно кода. Първо инициализираме скенера за четене на данни от конзолата, както и една променлива за числото,

въведено от оператора. Използваме **do-while** цикъл, в тялото на който искаме от потребителя да въведе число от 1 до 10. Тялото на цикъла се изпълнява задължително поне веднъж. Ако оператора е въвел правилно число между 1 и 10, условието за повтаряне на тялото няма да е изпълнено и програмата ще продължи своя ход. Ако операторът е въвел число извън този интервал, условието на цикъла ще е истина и тялото на цикъла ще се повтори. Тогава програмата ще изиска от оператора отново да въведе число между 1 и 10 и така докато операторът не въведе валидна стойност.

Цикъл „for“

For е третата конструкция за изпълнение на цикъл в Java и е най-широко използваната. **For** цикълът изглежда по следния начин:

```
for(инициализация; условие; контрол) {  
    тяло  
}
```

Конструкцията се състои от следните компоненти:

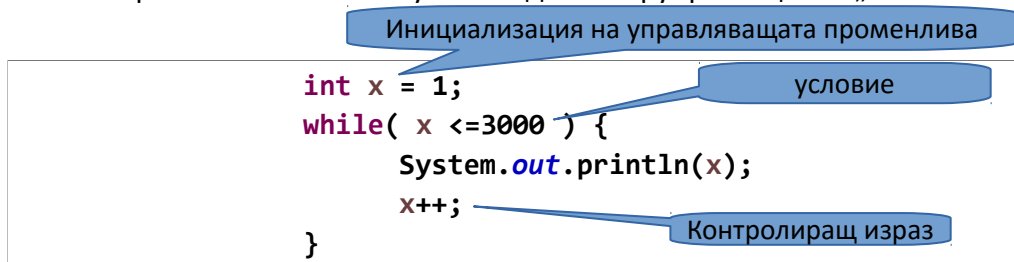
1. Ключовата дума **for** – по този начин компилаторът разбира, че следва конструкция за цикъл.
2. Инициализиращ израз – тук можем да инициализираме променлива, чрез която да управляваме цикъла.
3. Условие на цикъла – израз от булев тип, обикновено обвързан с променливата в инициализиращия блок. Условието на цикъла се проверява след всяка итерация, за да се определи дали цикълът трябва да изпълни тялото си отново.
4. Тяло на цикъла (**body**) – това е парчето код, което искаме да повтаряме.
5. Контролен израз – обвързан с промяна на стойността на променливата, контролираща цикъла.

For цикълът работи по следния начин: за (точка 1) променлива „x“ с начална стойност (точка 2). Докато (точка 3) е в сила някакво условие за

стойността на „x“, изпълни тялото на цикъла (точка 4), след което промени стойността на „x“ (точка 5) и се върни към проверка на условието (точка 3) и т.н.

Защо цикълът „for“ е най-често използван ?

Нека си припомним какво е нужно за да конструираме цикъл „while“.

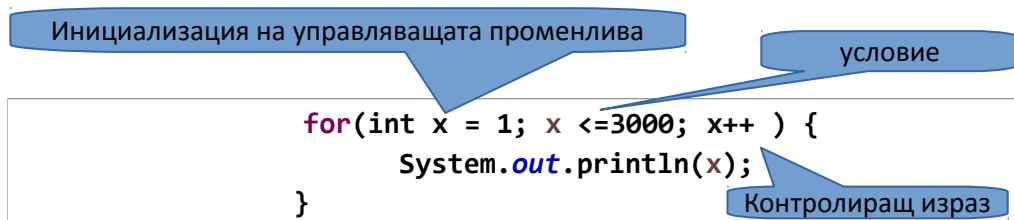


В посочения пример декларираме променлива `x` и я инициализираме със стойност 1. Този израз, обаче, в една реална програма може да не е непосредствено над цикъла, а по-нагоре в кода.

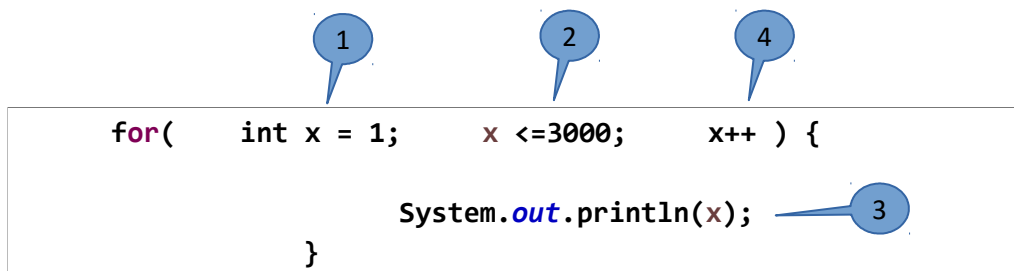
Също така тялото на цикъла може да не е 2 реда, както тук, а 20-30 реда. Тогава конструкцията „`x++`;“ няма да е толкова видима и кода става трудно четим, защото по-трудно човек ще се ориентира каква е идеята на цикъла.

Конструкцията „for“ събира всички компоненти, нужни за цикъла, на един ред – още при дефиницията му. По този начин много лесно можем да разберем каква е идеята на цикъла, без да се ровим в тялото му или да търсим инициализираната променлива някъде другаде в кода.

По този начин гореописаната конструкция може да се реализира с For цикъл ето така:

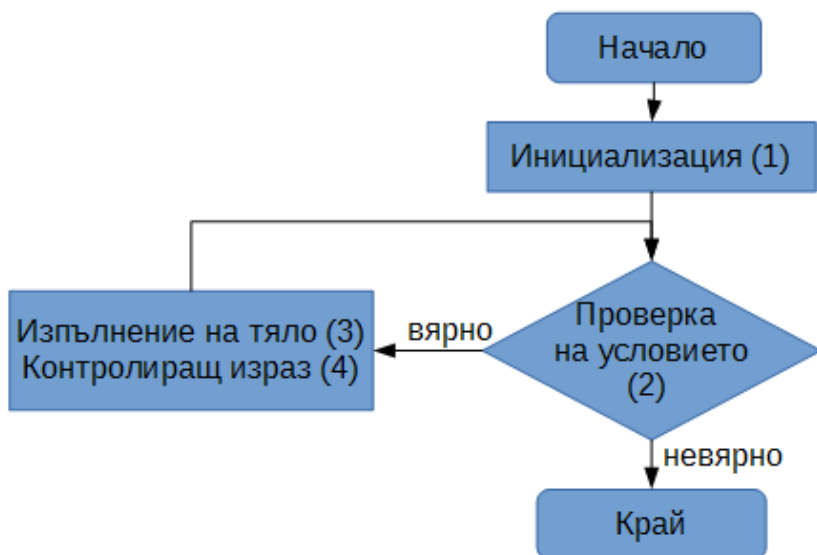


Важно е да се знае точната последователност от операции, които се извършват при изпълнение на „for“ цикъла:



Цикълът започва с инициализиране на управляващата променлива, в случая „x“ (1). След това се проверява условието за изпълнение на цикъла (2). Условието може да е невярно за началната стойност на „x“ - тогава тялото на цикълът няма да се изпълни и цикълът приключва. Ако условието е изпълнено, тялото на цикъла се изпълнява (3). След изпълнение на тялото на цикъла, се изпълнява контролиращият израз – в случая „x++“ (4). Той служи за промяна на стойността на управляващата променлива. След това отново се проверява условието и ако все още е в сила, тялото на цикъла се изпълнява отново. Тази последователност от проверка, изпълнение на тялото и изпълнение на контролиращия израз се върти докато условието за изпълнение на цикъла не стане невярно.

Следната блок схема визуализира работата на „for“ цикъла:



„For“ цикъл с няколко управляващи променливи

Възможно е да имаме няколко управляващи променливи за целите на цикъла. За да покажем това, нека решим следната задача: Да се напише програма, която визуализира две числа по следния начин:

1 – 10

2 – 9

3 – 8

....

9 – 2

10 – 1

От примера можем да видим, че първото число започва със стойност 1 и трябва да се увеличава с 1 докато не стигне стойност 10. Второто число трябва да започне със стойност 10 и да се намалява с 1 докато не стигне стойност 1.

За да реализираме задачата с „for“ цикъл, трябва да инициализираме две променливи съответно със стойности 1 и 10. Трябва първата променлива да се увеличава с 1, а втората да се намалява с 1 и така докато първата променлива не стане 10, а втората не стане 1.

Решението изглежда така:

```
for( int x = 1, y = 10; x <=10 && y >=1; x++, y-- ) {
    System.out.println(x + " - " + y);
}
```

Резултатът от изпълнението на цикъла е следния:

```
1 - 10
2 - 9
3 - 8
4 - 7
5 - 6
6 - 5
7 - 4
8 - 3
9 - 2
10 - 1
```

Безкраен „For“ цикъл

Начините за реализиране на безкраен „for“ цикъл са няколко:

1. Празен цикъл – можем да изпуснем изцяло инициализиращата част, условието и контролиращият израз. В такъв случай тялото на цикъла ще се изпълнява безкрайно.

```
for( ; ; ) {  
    System.out.println(" endless printing... ");  
}
```

2. Липса на контролиращ израз – тогава стойността на управляващата променлива няма да се променя и условието за изпълнение на цикъла винаги ще е вярно.

```
for(int x = 1; x < 5 ; ) {  
    System.out.println(x);  
}
```

3. Поставяне на условие, което винаги ще бъде в сила – това се получава, когато контролиращият израз няма ефект над условието

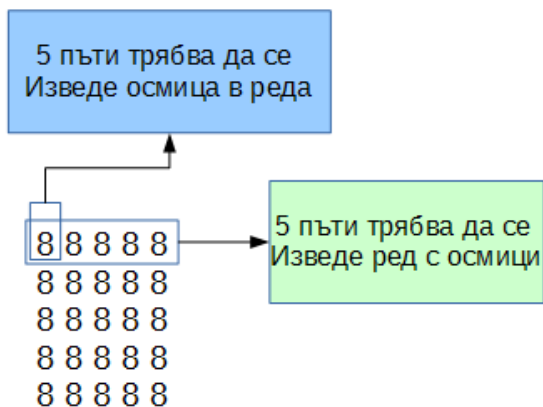
```
for(int x = 1; x < 5 ; x--) {  
    System.out.println(x);  
}
```

Вложени цикли

Досега разглеждахме примери, в които тялото на цикъла се състои от прости структури, като например да се изведе нещо на екрана. Понеже знаем, че в тялото на цикъла може да се съдържат всякакви конструкции, то това значи, че можем да имаме и конструкция на цикъл в тялото на друг цикъл.

Защо бихме ползвали вложен цикъл ? Нека се опитаме да решим следната задача: Да се изведе на екрана цифрата 8 в табличен вид – на 5 реда по 5 цифри на ред.

Желаният резултат е следният:



```
for(int col = 1; col <= 5 ; col++) { //за пет колони
    System.out.print("8"); //отпечатваме по една осмица
}
```

Резултатът от този цикъл е следният:

```
88888
```

```
for(int col = 1; col <= 5 ; col++) { //за пет колони
    System.out.print("8"); //отпечатваме по една осмица
}
for(int col = 1; col <= 5 ; col++) { //за пет колони
```

```

        System.out.print("8");//отпечатваме по една осмица
    }

```

Резултатът, обаче, този път ще е :



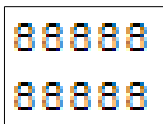
Кодът за изписване на два реда в конзолата сега би изглеждал така:

```

    for(int col = 1; col <= 5 ; col++) {//пет пъти
        System.out.print("8");//отпечатваме по една осмица
    }
    System.out.println();//слагаме нов ред
    for(int col = 1; col <= 5 ; col++) {//пет пъти
        System.out.print("8");//отпечатваме по една осмица
    }
    System.out.println();//слагаме нов ред

```

Резултатът вече е :



Забелязваме, че в последният вариант на решението, повтаряме код. Колко пъти трябва да го повторим, за да направим пет реда от осмици ? А ако ни трябваша 1000 реда от осмици ?

Ето защо тук е удачно да използваме цикъл, който да повтори конструкциите за изписване на ред от осмици в конзолата толкова пъти, колкото ни е нужно.

```

    for(int row = 1; row <= 5 ; row++) {
        for(int col = 1; col <= 5 ; col++) {
            System.out.print("8");

```

Един ред от осмици

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

Нов ред в конзолата

Нужно е след края на вътрешния цикъл да сложим конструкция за изписване на нов ред, за да може следващия ред с осмици да е под предишния.

Резултатът е следния:

```
88888  
88888  
88888  
88888  
88888
```

Ключова дума „break“

В практиката има случаи, в които по време на изпълнението на цикъл е нужно да прекратим цялата конструкция.

Дадена е например следната задача: Да се състави програма, която изписва всички числа от 1 до N, но когато срещне трицифрено число, програмата трябва да приключи изпълнение. Ако за N въведем число 2000, то тогава цикъла за извеждане би се завъртял 2000 пъти. На стотното завъртане, обаче, ще срещнем трицифрено число и цикъла трябва да прекрати работа.

За да постигнем този ефект, можем да използваме ключовата дума „**break**“. Чрез нея указваме, че искаме да прекратим изпълнението на цикъла. Тогава останалата част от тялото на цикъла няма да се изпълни и програмата ще продължи да изпълнява следващите конструкции след цикъла.

```

Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
for(int x = 1; x <= n ; x++) {
    if(x == 100) {
        break;
    }
    System.out.println(x);
}

```

При x = 100
цикъла приключва работа

Ключова дума „continue“

Нека решим следната задача : Да се напише програма, която визуализира всички числа от 1 до 100, но без числата между 70 и 80.

В тази задача е нужно при определени числа да не се изпълняват конструкциите от тялото на цикъла, а директно да се премине към следващата итерация. Тогава използваме ключовата дума „**continue**“. Чрез нея указваме, че искаме да прекратим текущото изпълнение на тялото на цикъла и да преминем към следващата итерация.



Използването на думата „continue“ не прекратява изпълнението на цикъла. Continue прекратява само текущата итерация и преминава към следващо завъртане на цикъла.

Решението на задачата би изглеждало така:

```

Scanner sc = new Scanner(System.in);
int x = sc.nextInt();
for(int x = 1; x <= 100 ; x++) {
    if(x >= 70 && x <= 80) {
        continue;
    }
    System.out.println(x);
}

```

При x между 70 и 80 останалата част от тялото на цикъла не се изпълнява.

Упражнения

1. Да се изведат в конзолата числата от -10 до 10.
2. Да се изведат в конзолата всички нечетни числа от 1 до 40.
3. Да се прочете число от конзолата и да се изведе сбора на всички числа от 1 до въведеното число.
4. Да се въведе число от конзолата и да се определи дали е просто. Просто число е това, което се дели САМО на 1 и на себе си.
5. Да се състави програма, която прочита две числа от конзолата – **p** и **q**. Да се изведе таблица с **p** реда и **q** колони, която да съдържа числа в следната последователност:

Пример: за **p** = 5 и **q** = 5:

| | | | | |
|----|----|----|----|----|
| 11 | 12 | 13 | 14 | 15 |
| 21 | 22 | 23 | 24 | 25 |
| 31 | 32 | 33 | 34 | 35 |
| 41 | 42 | 43 | 44 | 45 |
| 51 | 52 | 53 | 54 | 55 |

Глава 6. Масиви

В тази глава

Защо се нуждаем от масиви и какво представляват

Декларация и инициализация на масив

Достъп и обхождане на елементите на масив

Четене и отпечатване на масив от и на конзолата

Сравнение и копиране на масиви

Защо се нуждаем от масиви и какво представляват?

Нека имаме за задача да създадем програма, която да съхранява и да обновява оценките на група от 20 ученици. Това от което ще имаме нужда са 20 променливи, които да декларираме и инициализираме. Обновяването на тези оценки също няма да е лесно, защото за даден номер на ученик, ние ще трябва да намерим конкретната променлива, която пази неговата оценка и да променим нейната стойност. За да избегнем писането на толкова много код и съответно огромното повторение, което както вече знаем е лоша практика в програмирането, езикът Java ни дава механизъм за лесно създаване на много променливи наведнъж чрез конструкция, наречена **масив**. Също така променливите в масив ще могат лесно да бъдат достъпвани по техния пореден номер, който ще наричаме **индекс**.

Масивът представлява голям блок от памет, разделен на определен брой клетки, като във всяка клетка ние можем да съхраняваме една променлива. По този начин ние можем да достъпваме и модифицираме голям брой променливи много бързо.

Ограничения, които имаме при работата с масиви :

- всички елементи са от един и същ тип – цели числа, символи, булеви стойности и т.н.
- веднъж създаден, масивът не може да променя своя размер или с други думи броя променливи, които съдържа. Ако искаме по-голям или по-малък масив, трябва да създадем нов и да копираме елементите един по един от стария в новия масив.
- за да достъпим някой елемент от масив, е необходимо да знаем неговия **индекс**, тоест поредния номер на елемента. Особено то тук е, че елементите са индексирани (или номерирани) започвайки от **0**. Така ако имаме например масив с 6 елемента, то те ще бъдат индексирани от 0 до 5. Ето как изглежда в паметта, примерен масив с 6 елемента, попълнен с цели числа :

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|----|---|----|---|
| 5 | 10 | -2 | 1 | 19 | 1 |

Декларация и инициализация на масив

Създаването на масив с даден брой елементи и даден тип става със следната конструкция:

```
<тип на елементите в масива>[] <име на масива> = new <тип на  
елементите в масива>[<брой елементи>;
```

В първата част ние декларираме променлива от тип масив използвайки квадратни скоби ([]) след името на типа. Във втората част след знака "=" използваме оператора **new**, за да заделим необходимото количество памет за даден брой елементи от конкретен тип. Броят елементи трябва да бъде цяло, неотрицателно число. Нека да разгледаме няколко примера за създаване на масиви от определен тип и брой елементи в тях:

```
//крайните оценки на 20 ученика  
float[] studentScores = new float[20];  
//активните потребители в система с 300 потребители  
boolean[] activeUsers = new boolean[300];  
//примерен масив само с 1 елемент  
int[] testArray = new int[1];
```

В примерите от по-горе ние създаваме с един замах 20, 300 и 1 променливи от даден тип. Тези променливи ще имат за начална стойност 0.0, false и 0 съответно за променливите от първия, втория и третия масив. Както ще видим по-нататък, можем да зададем стойност на елементите, като ги достъпваме по индекс една по една. Ако предварително обаче знаем стойностите на променливите в един масив, то можем да използваме следната по-кратка конструкция, за да създадем и инициализираме масив :

```
//работните часове за 4-ма работника  
//забележете директното задаване на стойности в {}  
int[] workersHours = {8, 10, 6, 4};
```

Достъп и обхождане на елементите на масив

За да достъпим дадена променлива в масив, ние трябва да знаем нейният **индекс**. Използваме името на масива, следвано от квадратни скоби, като в тях поставяме индекса на елемента. След това можем да работим с този елемент, както работим с най-обикновена променлива – можем да присвоим стойност, да я изведем в конзолата, да я проверим с условен оператор и т.н. Ето как изглежда например конструкция, при която искаме да сменим стойността на даден елемент :

```
<име на масива>[индекс на елемента] = <нова стойност>;
```

Пример за създаване и използване на елементите на масив :

```
//създаваме масив с 3 елемента  
//представящ оценки на 3-ма ученика  
float studentScores[] = new float[3];  
//задаваме стойност за първия, втория и третия ученик  
//забележете, че номерирането на елементите започва от 0  
studentScores[0] = 23.2f;  
studentScores[1] = 13.2f;  
//можем да използваме елементите на масив, като най-  
обикновенни променливи и да конструираме всякакви изрази с тях  
studentScores[2] = studentScores[0] + studentScores[1];  
  
//номер на студент, за който искаме информация  
int choosenStudent = 1;  
//индексът също може да бъде стойност на целочислена  
променлива
```

```
System.out.println(studentScores[choosenStudent]);
```



Елементите на масива са индексирани от 0 до техният брой минус едно. Ако пробвате да достъпите елемент на невалидна позиция в масива, то програмата ще прекъсне и ще получите грешка в конзолата. Затова е хубаво когато използваме променливи в квадратните скоби, както в последния пример от по-горе с променливата "choosenStudent", да проверяваме с условен оператор дали стойността на тази променлива е валиден индекс в масива.

Обхождане на елементите на масив

Лесно можем да вземем елемент на произволна позиция в масив, знаем също така, че елементите са номерирани от 0 до техния брой минус 1, то тогава можем с цикъл, който брои в този интервал увеличавайки с 1, да обходим всички елементи на масив. Нека да разгледаме следния пример, представящ данни, за оставащите часове работа за група работници в началото на деня в дадена компания. Всички елементи на дадения масив се инициализират със стойност 8, колкото са и часовете работа в началото на работния ден.

```
//работни часове в началото на деня за 52-ма работници
int workHours[] = new int[52];
//обикаляме всички елементи с индекси от 0 до 51
//използвайки цикъл for
for (int workerIndex = 0; workerIndex <= 51; workerIndex++) {
    //за всеки индекс на работник
    //задаваме, че му остават 8 часа работа
    workHours[workerIndex] = 8;
}
```

Горният пример страда от един основен недостатък – какво ще стане ако броя на работниците в компанията се промени в даден момент, което често става с персонала на компаниите в днешно време? Ще е необходимо всеки път освен размера на масива, да променим и крайната позиция, до която се върти цикъла. За да направим кода по-лесен за промяна можем да вземем броя елементи на даден масив използвайки следната конструкция :

```
<име на масив>.length
```

Така цикълът от по-горе вече може да бъде преправен по следния начин:

```
//обикаляме всички елементи с индекси от 0 до последния  
валиден индекс в масива използвайки цикъл for  
  
for (int workerIndex = 0; workerIndex <= workHours.length-1;  
workerIndex++) {  
    //останалия код остава непроменен...
```

Четене и отпечатване на масив в конзолата

Продължавайки казаното от по-горе, щом можем да обиколим всички елементи на масив, то можем също така с лесна модификация да прочетем стойностите на всички елементи на масив от конзолата. Отново обикаляме с цикъл всички валидни позиции на елементи и за всеки елемент прочитаем неговата стойност от конзолата използвайки **Scanner** класа.

```
//нов масив с 6 елемента - комбинация от това  
int totoNumbers[] = new int[6];  
//Scanner за четене от конзолата  
Scanner sc = new Scanner(System.in);  
//за всеки валиден индекс на елемент в масива  
for (int index=0; index < totoNumbers.length; index++) {  
    //прочитаем стойността на всеки елемент с този индекс  
    totoNumbers[index] = sc.nextInt();
```

```
}
```

Аналогично можем и да изведем всички елементи на даден масив :

```
//нов масив с 6 елемента - комбинация от toto
int totoNumbers[] = {2, 23, 12, 42, 14, 5};
//за всеки валиден индекс
for (int index=0; index < totoNumbers.length; index++) {
    //отпечатваме стойността на всеки елемент с този индекс
    System.out.println(totoNumbers[index]);
}
```

Копиране и сравняване на масиви

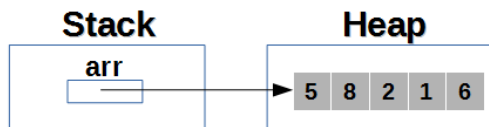
Типът масив не е примитивен тип данни. Това означава, че масивите се съхраняват по различен начин в паметта и операциите по копиране и сравняване се държат по по-различен от очаквания. За да разберем това първо ще трябва да разгледаме как се разполагат масивите в паметта, а също така и да разгледаме разликите между двата основни вида памет – **Стек (Stack)** и **Хийп (Heap)**.

Разликата между Стек (Stack) и Хийп (Heap)

Накратко ще обясним, че стекът е част от паметта, която се заделя за една програма, за да съхранява своите променливи. Тя е бърза памет, но малка като размер. От друга страна хийпът е по-голяма памет, но достъпът до нея е по-бавен. Това, което трябва да запомним, е че променливите от примитивен тип, заедно с техните стойности, се съхраняват директно в стека. От друга страна променливите от референтните типове (масиви, обекти и др.) в стека съхраняват **адреса** или **референция** към тези данни. Самите елементи на масива вече се съхранява в хийпа. Тоест ако изпълним следния ред от код :

```
int[] arr = {5,8,2,1,6};
```

то паметта ще изглежда по следния начин :



```
System.out.println(arr);
// в конзолата ще видим нещо такова: [I@15db9742
```

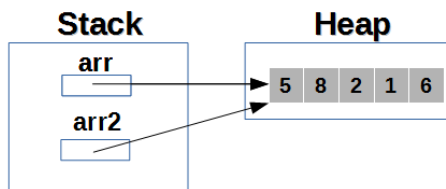
Как работят операторите "=" и "==" при референтни типове данни?

Когато искаме да запишем един масив в друг, използвайки оператора "=", това което се случва е не да се създаде втори масив в хийпа със същите елементи. Единственото, което получаваме като резултат, е втора референтна променлива в стека, която сочи към същият масив в хийпа. С други думи – имаме един и същ масив, сочен от две променливи в стека.

Ето пример за това :

```
int arr[] = {5,8,2,1,6};
int arr2[] = arr; //тук не създаваме нов масив,а просто сочим
към същия
```

В този момент паметта би изглеждала по следния начин :



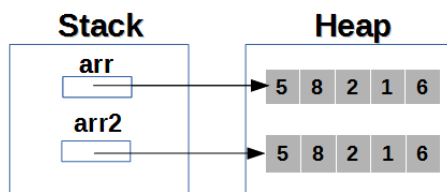
Какво ще означава това за нас? Че промяната по единия масив веднага ще промени и другия, тъй като и двете променливи (референции) сочат към една и съща област в паметта. Т.е. ние реално не правим копие, а просто можем да модифицираме и четем едни и същи данни по два начина. Например :

```
int arr[] = {5,8,2,1,6};  
int arr2[] = arr; //тук не създаваме нов масив, а просто сочим  
към същия  
arr2[1] = 3; //модифицираме използвайки референцията arr2  
System.out.println(arr[1]); //тук ще получим 3, защото  
оригиналния масив ще бъде променен от предишния ред
```

Какво трябва да направим ако наистина искаме да създадем копие на нашия оригинален масив? За тази цел ще трябва да заделим памет за нов масив със същата големина като оригиналния и след това, обикаляйки елементите един по един, да ги копираме в новия масив. Така вече ще направим истинско копие на оригиналния масив и промените по единия масив няма да рефлектират върху другия.

```
int arr[] = {5,8,2,1,6};  
int arr2[] = new int[arr.length]; //тук създаваме нов масив, с  
големината на оригиналния  
for (int index=0;index<arr.length; index++) {  
    //за всеки индекс копираме стойността на елемента в  
новия масив  
    arr2[index] = arr[index];  
}  
arr2[1] = 3; //модифицираме копието използвайки arr2  
System.out.println(arr[1]); //оригиналния масив вече ще си  
остане непроменен
```

В примера от по-горе създаваме чисто нов масив и копираме всички елементи един по един. По този начин вече в паметта ще имаме две различни референции към два различни масива :



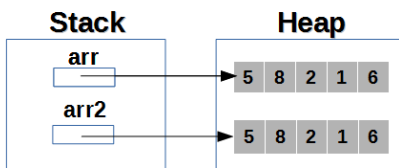
Как стоят нещата със сравняването на масиви? Както вече видяхме, масивите са референтни типове и сравняването на два масива с оператора "==" всъщност единствено ще ни покаже дали двете референции сочат една и съща памет, което не е точно това, което бихме искали да проверим. Ето пример за това как за масиви с едни и същи данни и размер, излиза че те са различни :

```
int arr[] = {5,8,2,1,6}; //създаваме два еднакви масива
int arr2[] = {5,8,2,1,6};
if (arr==arr2) {
    System.out.println("Масивите са еднакви");
}
else {
    System.out.println("Масивите не са еднакви");
}
```

Резултатът обаче ще бъде, че масивите няма да бъдат еднакви :

Масивите не са еднакви

Както и оператора "=" така и оператора "==" работи с референциите, а не със самите данни. В този момент паметта изглежда по следния начин :



Референциите "arr" и "arr2" сочат различни адреси в паметта и поради тази причина резултатът е, че масивите са различни. Можем да сравним 2 произволни масива за равенство по следния начин :

```
if (arr.length != arr2.length) { //проверяваме първо размерите
на двата масива
    System.out.println("Масивите са различни.");
}
else { //ако масивите са с еднакви размери
    int index=0;
    for (index=0; index < arr.length; index++) {
//проверяваме всички елементи
        if (arr[index] != arr2[index]) { //ако открием
различие извеждаме, че са различни и прекратяваме цикъла
            System.out.println("Масивите са различни.");
            break;
        }
    }
    //ако цикъла е стигнал до края си и не е бил
прекъснат,заради различие на елементи, извеждаме че масивите
са еднакви
    if (index >= arr.length) {
        System.out.println("Масивите са еднакви.");
    }
}
```

Упражнения

- Прочетете масив от екрана с 10 елемента, числа с плаваща запетая и изведете сумата на елементите в масива.
- По дадени два масива да се провери дали първият, обърнат наобратно, е равен с втория.
- Даден е масив с 20 елемента, цели числа. Да се изведе най-голямото число в масива.
- По даден масив със 15 символа, променливи от тип **char**, да се провери дали някой от тези символи е малка латинска буква.

Упътвания

- След прочитането на масива, обиколете елементите му с цикъл и в нова, предварително инициализирана променлива добавяйте всички елементи на масива. Накрая изкарайте тази променлива като резултат.
- Прегледайте индексите на елементите, които трябва да сравнявате – 0-вият от първия трябва да се сравни с последния от втория масив, 1-вия с предпоследния и т.н.
- Създайте променлива `max` с много малка отрицателна стойност. Обиколете елементите на масива и ако някой елемент е по-голям от променливата `max` – запишете тази стойност в нея. Накрая изведете `max`.
- Обиколете масива и проверете дали всеки символ е между символите 'a' и 'z'. Ако намерите такъв изведете съобщение и прекъснете цикъла. След цикъла проверете дали той е завършил нормално изпълнението си. За повече информация погледнете сравнението на два масива за равенство.

Глава 7. Двумерни масиви

В тази глава

Какво са двумерните масиви

Връзка с едномерните масиви

Създаване и инициализация

Достъп до елементите на двумерен масив

Обхождане, четене и извеждане на двумерни масиви

Сума на елементите, минимален и максимален елемент в двумерен масив

Какво са двумерните масиви

Досега разглеждахме масиви с едно измерение, които изглеждаха като таблица с един ред и много колони. Много често представяме, обаче, данните под формата на таблица с много редове и колони. Такова представяне на данните би изисквало от нас за всеки ред да имаме едномерен масив с брой клетки, равен на броя колони, които са ни нужни. Това означава, че трябва да създаваме много на брой едномерни масиви, които да използваме по много сходен начин. В предната глава видяхме как масивите решават проблема със създаването на много променливи от един тип и сходно приложение. Сега, обаче, имаме проблем с много масиви със сходно приложение. Можем да си решим и този проблем, като отново сложим масивите в масив. По този начин ще конструираме т.н. **двумерен масив**. Двумерните масиви често се наричат **матрици**. Можем да си представим двумерния масив по следния начин :

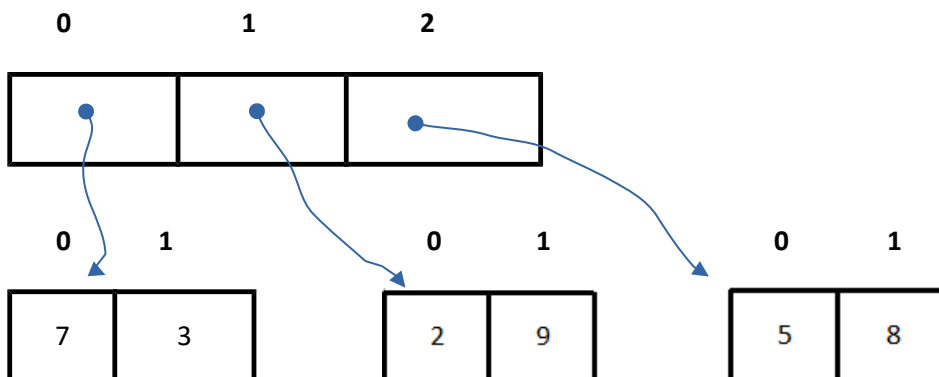
| индекси | 0 | 1 |
|---------|---|---|
| 0 | 7 | 3 |
| 1 | 2 | 9 |
| 2 | 5 | 8 |

За по-нагледно представяне, двумерните масиви се визуализират като таблица, в която редовете всъщност са клетките на едномерен масив, в който има едномерни масиви. Клетките на вътрешните масиви са дефакто колоните на таблицата. Затова често ще използваме термините **ред** и **колона** като идентификатори на клетки от двумерния масив.

Връзка с едномерните масиви

Казахме, че елементите на масив могат да бъдат от всякакъв тип, стига всички елементи да бъдат от един и същи такъв. Двумерният масив всъщност е едномерен масив, елементите на който са също едномерни

масиви. В паметта двумерният масив от по-горе ще изглежда по следния начин :



Създаване и инициализация

Създаването на двумерен масив не се различава особено от това на едномерния – необходимо е просто да зададем броя колони и броя редове :

```
// 5 книги с 30 рейтинга за всяка от тях  
int[][] bookRatings = new int[5][30];
```

Както и в едномерния масив, така и в двумерен масив можем да зададем предварително стойности за всички елементи. Тъй като двумерния масив е масив от масиви, то трябва за всеки ред да зададем стойност и да представим това като поредица от инициализирани едномерни масиви, разделени със запетая. Ето пример:

```
// 3 работника с 5 оценки за техните умения  
int workerRatings[][] = {  
    {6, 3, 8, 8, 9},  
    {1, 2, 5, 2, 2},  
    {8, 10, 10, 10, 9}  
};
```

Достъп до елементите на двумерен масив

Четенето и промяната на елементите на двумерен масив се получава като използваме името на масива и след това в две двойки квадратни скоби посочим първо реда, а след това и колоната на елемента, който искаме да модифицираме или прочетем. Нека например да вземем масива от по-горе, в който съхраняваме оценки по някакви критерии за група работници. Ще променим данните за третия работник. Припомняме, че както и при едномерни масиви и тук номерацията на редове и колони започва от 0.

```
//задаваме оценка по 2-рия критерий за 3-тия работник
workerRatings[2][1] = 1;
```

Обхождане, четене и извеждане на двумерни масиви

За да обходим последователно всички елементи на двумерен масив, ще са ни необходими два вложени цикъла – първият, който да премине по всички редове, а вторият да обходи за всеки ред всички колони. Можем да получим броя редове като прочетем свойството `.length` на масива. Ако масива е матрица, т.е. всички всички редове са с еднакъв брой колони, то тогава можем да намерим броя колони като вземем 0-вия елемент на масива (първия ред) и вземем неговото `.length` свойство. Така, имайки броя редове и колони и знаейки, че номерацията им започва от 0, можем да изведем всички елементи на масива по следния начин :

```
//за всички редове
for (int row = 0; row < workerRatings.length; row++) {
    //за всички колони в този ред
    for (int col = 0; col < workerRatings[0].length; col++)
    {
        //ще изведем текст заедно с текущите индекси и стойността на
        текущия елемент
        System.out.println("Оценка по критерий " + col + " за
        работник с номер " + row + " : " + workerRatings[row][col]);
    }
}
```


Следвайки този алгоритъм, можем също така и да прочетем елементите на двумерен масив от конзолата. Отново ще обиколим всички елементи и за всеки елемент ще прочетем съответната стойност използвайки **Scanner**. За по-голяма яснота преди това ще изведем съобщение за да е ясно елемент на кой ред и колона въвеждаме в момента :

```
Scanner sc = new Scanner(System.in);
//нов масив с данни - 2 реда по 3 колони
int data[][] = new int[2][3];
//обхождаме всички клетки
for (int row = 0; row < data.length; row++) {
    for (int col = 0; col < data[0].length; col++) {
        //подканващо съобщение към потребителя
        System.out.println("Въведете данни за ред : " +
                            row + " колона : " + col + ":");
        //прочитаме стойността в съответната клетка в масива
        data[row][col] = sc.nextInt();
    }
}
```

Сума на елементите, минимален и максимален елемент в двумерен масив

Следвайки логиката от по-горе можем лесно да намерим сумата на елементите, най-големия и най-малкия елемент. Достатъчно е да си създадем предварително три променливи, които да съхраняват търсените резултати. Преди да започнем проверката на всички елементи, ще присъединим много ниска стойност за променливата, в която търсим най-големият елемент. Правим това, защото всеки път, когато намерим по-голяма стойност, ще я записваме в тази променлива. Аналогично в променливата, в която ще търсим най-малкия елемент записваме достатъчно голяма стойност. Последно инициализираме сумата с 0 и всеки път добавяме поредния елемент, който обхождаме.

```

//в тези променливи съхраняваме
//най-малката, най-голямата стойност
//и сумата на елементите
int max = -99999;
int min = 99999;
int sum = 0;

//обхождаме всички елементи
for (int row = 0; row < data.length; row++) {
    for (int col = 0; col < data[0].length; col++) {
        //добавяме всеки елемент към общата сума
        sum += data[row][col];
        //ако намерим елемент който е по-голям от най-
големия до момента го записваме в променливата max
        if (data[row][col] > max) {
            max = data[row][col];
        }
        //ако намерим елемент който е по-малък от най-
големия до момента го записваме в променливата min
        if (data[row][col] < min) {
            min = data[row][col];
        }
    }
}
System.out.println("Най-малък елемент : " + min);
System.out.println("Най-голям елемент : " + max);
System.out.println("Сума на елементите : " + sum);

```

Какво ще стане, обаче, ако всички елементи в масива са по-големи от предварително заложената ни минимална стойност или по-малки от предварително заложената ни максимална ? Тогава стойностите на max или min няма да се променят и програмата няма да работи добре.

Вместо да слагаме на променливите min и max съответно произволно ниски и високи стойности, можем да заложим за тяхна стойност - тази на първият елемент в двумерния масив. По този начин винаги резултатът ще бъде правилен.

Упражнения

- По дадена матрица от цели числа, изведете числото, което се среща най-често. Ако има повече от едно такова – изведете първото срещнато.
- По даден квадратен двумерен масив от естествени числа, да се отпечатаат диагоналите на масива.
- Имате предварително въведени стойности на елементи в двумерен масив от естествени числа. Да се състави програма, чрез която се извеждат стойностите на елементите в двумерен масив след обръщането му на +90 градуса.

Пример:

1,2,3,4

5,6,7,8

9,10,11,12

13,14,15,16

Изход:

13,9,5,1

14,10,6,2

15,11,7,3

16,12,8,4

Упътвания

– За всеки елемент обиколете масива, за да преброите колко пъти се среща този елемент. Ако текущата бройка е по-голяма от максималната до момента – запишете текущото число и текущия брой като максимални за момента.

- Наблюдавайте индексите на съответните елементи и намерете зависимост. Необходим ви е по един цикъл за всеки от диагоналите.
- Наблюдавайте как се променят индексите на елементите, когато се извеждат по този начин. Използвайте обикновено извеждане, но добавете съответния израз при извеждането.

Глава 8. Методи. Рекурсия.

В тази глава

Защо повтарянето на код е лоша практика

Въвеждане в методите

Декларация и имплементация на метод

Извикване на метод

Пример за програма с методи

Област на видимост на променливите

Предаване на параметри към метод

Рекурсия

Упражнения

Защо повтарянето на код е лоша практика

В глава 5, в която разгледахме циклите, обяснихме как повтарянето на код може да доведе до трудна поддръжка и разчитане, както и до висок риск от грешки при редакция.

Циклите, обаче, не са универсално решение на повторението на код. Те ни вършат работа само в случаите, когато парче код трябва да се повтори няколко пъти в дадена ситуация. Но какво става, ако имаме повторение на едно парче код на различни места в приложението ?

Нека вземем предвид следния проблем: Имаме сайт за кредити, който предлага различни по параметри оферти за кредит – да кажем 30 кредита общо. Десет от кредитите са таргетирани само за потребители между 18 и 28 години, а останалите са за по-възрастни потребители. В такъв случай когато човек влезе на сайта, ще види 30 бутона за кредити, но при натискането на десет от тях, ще трябва да има валидация на годините му – дали са между 18 и 28. Тоест на десет различни места в системата ние трябва да напишем един и същ код за валидация на годините на потребителя.

```
if( age > 18 && age < 28) {  
    System.out.println("You can enter.");  
} else {  
    System.out.println("You are not allowed.");  
}
```

Какво ще стане, ако утре кредита се промени и вместо 18 годишни, минималния праг стане за 21 годишни ? Тогава на десет различни места ще трябва да променим условието. Това е предпоставка за грешки, защото дори да пропуснем едно от тези места, това означава, че хора, по-малки от 21 години, ще могат да се възползват от кредита.

Тук на помощ идват методите.

Въведение в методите

Методите са конструкции, които съдържат блок от код, изпълняващ точно определена операция. Идеята на методите е да се използват за решаването на даден конкретен проблем в нашата програма.

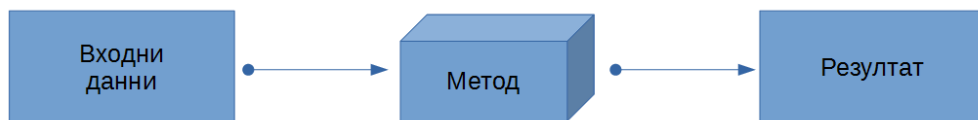
По-горе представихме един проблем – как да валидираме дали годините на потребителя са в определена граница. За да решим проблема, ни трябва входни данни – годините на потребителя. Трябва да изпълним някакви операции, за да разрешим проблема и трябва да представим някакъв резултат, който е решението на проблема.

Например ако годините на потребителя не са в нужната граница, трябва да укажем, че не е позволено той да вземе кредита. Ако годините попадат в границите, трябва да укажем, че потребителят може да вземе кредита.

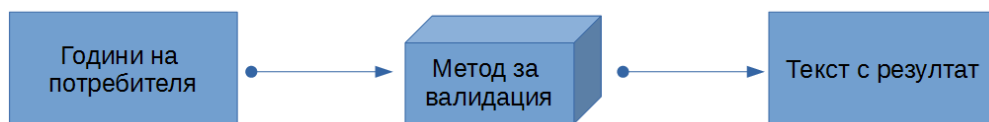
Съставните части на решаването на проблема са:

- входни данни;
- операции с входните данни;
- получаване на резултат.

Точно затова са конструирани и методите. Те представляват своеобразни конструкции, които решават даден проблем. За целта приемат входни данни и връщат резултат.



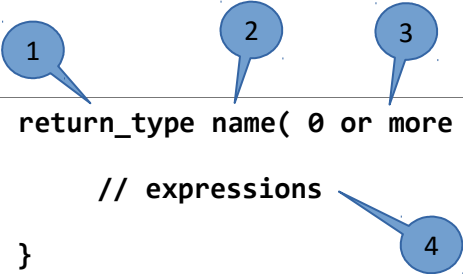
Ако вземем нашия пример по-горе, схемата би изглеждала така:



Декларация и имплементация на метод

Методите се състоят от следните съставни части:

- Тип на връщаната стойност – указва какъв тип стойност ще върне метода като краен резултат.
- Наименование – наименование на метода. По този начин можем да различаваме методите един от друг. Също така според наименованието на метода много лесно можем да се ориентираме какъв проблем решава той.
- Параметри – списък с променливи, които играят ролята на входни данни за проблема, който метода решава.



```
return_type name( 0 or more parameters) {  
    // expressions  
}
```

The diagram illustrates the components of a method signature with numbered callouts: 1 points to the return type, 2 points to the method name, 3 points to the parameters in parentheses, and 4 points to the opening curly brace.



Методите могат да имат 0 или повече параметри. Тоест един метод може да се декларира и инициализира без параметри, с един или с много параметри.

- Тяло – това е блок от код, в който се решава проблема, за който е създаден метода.



Методите могат и да не връщат резултат. В такива случаи за тип на връщаната стойност се записва ключовата дума „void“. Тя указва, че методът няма да върне резултат.

Нека вземем за пример проблема с кредитите. За да декларираме метод за валидация, бихме написали следното:


```

    1 static void validate(int age) {
    2     if( age > 18 && age < 28) {
    3         System.out.println("You can enter.");
    4     } else {
    5         System.out.println("You are not allowed.");
    6     }
    7 }

```



Забележете, че в примера при декларацията на метода има и ключовата дума „static“. Нейното значение и употреба няма да разглеждаме сега, а само ще споменем, че за да можете да изпробвате примерите чрез main метода на вашия проект, тази дума „static“ е нужна пред всеки метод.

Декларацията на метод винаги започва с указване на типа на връщаната стойност от метода ①. Типа може да е всеки познат тип – int, boolean, short и т.н. Ако метода няма да връща никакъв резултат, тогава за тип записваме ключовата дума „void“. След типа на връщаната стойност, следва наименованието на метода ②. Прието е наименованието на методите да започва с малка буква, като за сложни съчетания от думи, всяка следваща съставяща дума започва с главна буква. След наименованието следват отваряща и затваряща скоби ③. В скобите може да се добавят променливи, които да послужат за входни данни на метода. Променливите само се декларират, тоест записват се с тип и наименование.

Записването на тяло на метод наричаме „имплементация“. Тялото на метода представлява блок от код, който следва след декларацията му ④.



Добра практика е наименованието на методите да е глагол, който описва действието, извършвано от метода. По този начин кода става по-четим и лесно може да се разбере идеята на метода, без да се обръща внимание на тялото му.

Връщане на стойност от метод

За да укажем, че искаме да върнем стойност от метод, използваме ключовата дума „**return**“, след което записваме стойността, която искаме да върнем.

По този начин можем да използваме методи, които да правят сложни изчисления и директно да ни връщат резултата. Така можем да ползваме методите в изрази.

За да покажем тази функционалност, нека направим метод, който изчислява лицето на квадрат.

Какво е нужно, за да изчислим лице на квадрат ? Като входни данни ни трябва дължината на страна от квадрата. Като резултат ще върнем стойността на лицето на квадрата. В тялото на метода (операциите, които са нужни, за да решим проблема) ще умножим страната на квадрат, за да може да получим лицето.

Указваме, че резултата ще е число

Наименование, описващо действието в метода

```
static int calcSizeOfSquare(int side) {  
    int size = side*side;  
    return size;  
}
```

Страната на квадрата като входен параметър

Връщаме като резултат полученото лице на квадрата

Спомняте ли си, когато в първа глава на книгата описахме как да стартираме първата си програма на Java ? Тогава използвахме една конструкция, в която и досега записваме всички наши примери. Конструкцията изглеждаше така:

```
public static void main(String[] args) {  
    //кода на нашата програма  
}
```

Тогава не споменахме какво точно представлява този код, само взехме за даденост, че на мястото на коментара трябва да записваме нашите инструкции, а програмата ще започва изпълнението си от тях.

Всъщност тази конструкция представлява един обикновен метод с наименование „**main**“, входни данни **String[] args** – масив от символни низове (за тях ще научим в следващата глава) и тип на връщаната стойност – **void** – тоест метода не връща никаква стойност.

Методът **main** представлява стартовата точка на всяка една програма. Тоест когато стартирате вашата програма, първата инструкция, която ще се изпълни, ще бъде в началото на тялото на **main** метода.

Методът **main** се декларира в т.н. клас. Класовете няма да бъдат разгледани в настоящата книга, но за тях е достатъчно да знаем, че са по-голяма единица от код, която се съхранява в отделен файл.

Декларирането на методи, също както и методът **main**, може да се осъществи само в рамките на класа.



**Не можем да декларираме метод в тялото на друг метод.
Методи се декларираат само в рамките на класа.**

Ако погледнем по-отгоре на нещата, нашата програма с деклариран метод за изчисление на лицето на квадрат би изглеждала така:

```
public class Square {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
  
    static int calcSizeOfSquare(int side) {  
        int size = side*side;  
        return size;  
    }  
}
```

Методът **main**, от който започва
нашата програма

Методът **calcSizeOfSquare**,
Който изчислява лице на
квадрат

Какво ще се случи, когато стартираме гореописаната програма ? Отговорът е – **нищо**! Но защо ? След като имаме метод, който смята лицето на квадрат, защо този код не се изпълнява ?

По дефиниция кодът в методите не се изпълнява, докато програмистът не реши. За да изпълним кода на един метод, трябва да го „извикаме“.

Извикване на метод

Извикването на метод става чрез записването на името на метода, последвано от отваряща и затваряща скоби. Инструкцията за извикване на метод означава, че искаме да изпълним всичко, което се съдържа в тялото на този метод.



Ако при декларацията на метод сме подали списък с променливи в скобите за входни данни, то задължително при извикването на метода трябва да подадем в скобите стойности за тези параметри.

За да извикаме метода за изчисляване лицето на квадрат, трябва да изпишем наименованието на метода, след което в скоби да поставим стойности за входни данни. Тези стойности се наричат „аргументи“ на метода.

```
public class Square {
```

```
    public static void main(String[] args) {  
        int result = calcSizeOfSquare(5);  
        System.out.println(result);  
    }
```

Извикваме метода
с аргумент - 5

```
    static int calcSizeOfSquare(int side) {  
        int size = side*side;  
        return size;  
    }
```

Параметърът size приема
стойност 5.

В гореописания код извикваме метода за изчисляване лицето на квадрат, като изписваме наименованието му, а в скобите слагаме за аргумент стойността 5. Какво се случва при изпълнение на програмата ? При извикването на метода се стартира неговото тяло. Преди това, обаче, параметрите на метода получават стойности, които са подадени като аргументи при извикването. Тоест параметърът **int side** ще бъде инициализиран със стойност 5. След това тялото на метода ще се изпълни, като променливата **size** ще бъде инициализирана с лицето на квадрата, а именно **side*side = 5*5 = 25**. Стойността 25 ще бъде върната от метода чрез ключовата дума **return**. Върнатата стойност ще бъде присвоена на променливата **result** в тялото на **main** метода, след което ще бъде изписана в конзолата.

Когато имаме методи, които не връщат стойност, можем да ги извикваме като просто изпишем името им и им подадем аргументи.

Нека например извикаме метод за валидация на годините на потребителя в сайта за кредити:

```
public class Credit {

    public static void main(String[] args) {
        int age = 30;
        validateAge(age);
    }

    static void validateAge(int years) {
        if(years > 18 && years < 28) {
            System.out.println("Можете да вземете
кредит");
        } else {
            System.out.println("Не отговаряте на
условията");
        }
    }
}
```

Програмата ще извика метода **validateAge** с аргумент – стойността на променливата **age**. Ще се стартира тялото на метода, като неговия параметър **years** ще се инициализира със стойността на подадения аргумент **age**. След това ще бъде изпълнена условната конструкция, която проверява за валидност на годините и метода ще прекрати работа.

Важно е да се отбележи реда на изпълнение на конструкциите, когато се ползват методи. Нека проследим следния код:

```
public class Squares {  
  
    public static void main(String[] args) {  
        1 System.out.println("Начало на програмата");  
        2 System.out.println("Лице на квадрат със страна 5:");  
        3 calcSquareSize(5);  
        5 System.out.println("Лице на квадрат със страна 10:");  
        6 calcSquareSize(10);  
        8 System.out.println("Лице на квадрат със страна 99:");  
        9 calcSquareSize(99);  
        11 System.out.println("Край на програмата");  
    }  
  
    static void calcSquareSize(int side) {  
        4 7 10 System.out.println(side*side);  
    }  
}
```

Инструкциите в **main** метода се изпълняват една след друга отгоре надолу. Когато, обаче, се стигне до извикване на метод, следващата инструкция вече е в тялото на този метод. Когато всички инструкции от тялото на метода се изпълнят, се „излиза“ от метода и изпълнението се прехвърля към **следващата инструкция след реда на извикване на метода**.

Това е показано на примера по-горе. Първо се изпълняват инструкциите за начало на програмата ① и ②. След това следва извикване на метод - ③. Изпълнява се тялото на метода ④ със стойност на параметъра side = 5. Тялото на метода приключва и изпълнението се връща към инструкция ⑤. Следва отново извикване на метода с аргумент 10 - ⑥. Отново се изпълнява тялото на метода за стойност на side = 10 ⑦. Изпълнението на метода отново приключва и се стига до инструкция ⑧. Отново викаме метода с аргумент 99 ⑨. Тялото се изпълнява ⑩, след което се изпълнява инструкцията за край ⑪.

Резултатът от задачата би изглеждал така:

```
Начало на програмата  
Лице на квадрат със страна 5:  
25  
Лице на квадрат със страна 10:  
100  
Лице на квадрат със страна 99:  
9801  
Край на програмата
```

Пример за програма с методи

Нека решим следната задача: Дадено е число, въведено от потребителя. Трябва да валидираме дали числото отговаря на следните критерии:

- числото е положително;
- числото се дели на 3;

Ако числото отговаря на критериите, трябва да изведем на екрана както него, така и неговия квадрат и куб. Квадрат наричаме стойността, която се получава, когато повдигнем числото на втора степен, а куба е повдигнатото число на трета степен.

Ако числото не отговаря на условията, трябва да покажем текст за грешка.

Първото нещо, което трябва да направим, е да разделим задачата на подзадачи, всяка от които отговаря за един прост проблем. Имаме няколко ключови момента тук.

1. Валидиране на число по някакви критерии.
2. Извеждане на екрана на информация за число.
3. Извеждане на стандартен текст за грешка.

Понеже всеки един от тези ключови моменти е независим от другите и е сравнително прост, можем да го отделим в отделен метод. Това значи, че за целите на изпълнение на програмата са ни нужни три метода:

1. Метод, който приема за параметър число и връща булева стойност в зависимост от това дали числото отговаря на определени критерии;
2. Метод, който приема за параметър число и изписва в конзолата информация за него (стойността на числото, неговия квадрат и неговия куб);
3. Метод, който не приема никакви параметри и изписва стандартен текст за грешка в конзолата.

Първият метод изглежда така:

```
static boolean isValid(int num) {  
    boolean valid = (num > 0) && (num % 3 == 0);  
    return valid;  
}
```

Вторият метод изглежда така:

```
static void showNumberInfo(int num) {  
    System.out.println("Числото е" + num);  
    System.out.println("Неговия квадрат е" + num*num);  
    System.out.println("Неговия куб е" + num*num*num);  
}
```

Третият метод изглежда така:

```
static void showError() {  
    System.out.println("Невалидно число!");  
}
```

Нека помислим сега как трябва да навържем методите в нашата програма така, че тя да отговаря на изискванията. Трябва да прочетем число от конзолата и да проверим дали то отговаря на критериите. Това означава, че в условна конструкция **if** трябва да извикаме метода **isValid**,

като за аргумент подадем стойността на въведеното число. Според резултата, който метода върне, ще решим какво да правим. Ако стойността е true, ще извикаме метода за визуализиране на информация за числото. Ако стойността е false, ще извикаме метода за грешка.

```
Scanner sc = new Scanner(System.in);
int number = sc.nextInt();

if(isValid(number)) {
    showNumberInfo(number);
} else {
    showError();
}
```

Нека сега сгложим всички части на нашата програма:

```
public class Number {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        int number = sc.nextInt();

        if(isValid(number)) {
            showNumberInfo(number);
        } else {
            showError();
        }
    }

    static boolean isValid(int num) {
        boolean isValid = (num > 0) && (num%3 == 0);
        return isValid;
    }

    static void showNumberInfo(int num) {
        System.out.println("Числото е" + num);
    }
}
```

```
        System.out.println("Неговия квадрат е" +  
num*num);  
        System.out.println("Неговия куб е" +  
num*num*num);  
    }  
  
    static void showError() {  
        System.out.println("Невалидно число!");  
    }  
}
```

Област на видимост на променливите

Област на видимост на променлива наричаме мястото, където дадена променлива може да се използва. Всяка променлива има област на видимост, определяща се от това къде е декларирана.

Най-общо казано областта на видимост на една променлива е блока от код, в който тя е декларирана. Променливата може да се използва в инструкции, намиращи се след нейната декларация, както и в други вложени блокове от код, отново намиращи се след нейната декларация. Променливата може да се използва докато не се стигне края на блока от код, в който е декларирана.

Когато декларираме една променлива в нашия main метод, ние можем да използваме тази променлива само в тялото на този метод. Това е така, понеже тялото на един метод представлява на свой ред блок от код. Тогава казваме, че областта на видимост на променливата е тялото на метода.

Досега споменахме, че при дефиниране на метод можем да подадем списък с параметри, които да играят ролята на входни данни. Параметрите се подават под формата на обикновени променливи, които изписваме в скобите на метода.

```
static boolean isValid(int num) {  
    //тяло на метода  
}
```

Деклариран параметър

Важно е да се знае, че областта на видимост за параметрите на метода също се свежда до тялото на метода.



Променлива, декларирана в блок от код, може да се използва и във вътрешни блокове от код, но никога извън обхвата на блока, в който е декларирана.

Нека разгледаме следния метод:

```
static boolean isValid(int num) {  
    int x = 4;  
    if(x < 55) {  
        int y = x + num + 5;  
    }  
    System.out.println(x);  
    System.out.println(num);  
    System.out.println(y);  
}
```

Област на видимост на `x` и `num`

Област на видимост на `y`

Нямаме достъп до `y` тук

Имаме променлива `num`, която е параметър на метода. Имаме също така променлива `x`, декларирана в тялото на метода. Областта на видимост и на двете променливи представлява цялото тяло на метода, заедно с всички вложени блокове от код. Тоест `x` и `num` могат да бъдат използвани свободно както в блока на `if` конструкцията, така и навсякъде другаде.

Забележете, че във вложения блок на `if` конструкцията декларирахме още една променлива – `y`. Нейния обхват на видимост, обаче, представлява единствено блока от код, в който е декларирана, а именно – блока на `if`-а. Ако се опитаме да използваме `y` извън блока от код на `if` конструкцията, това няма да ни е позволено.

Предаване на параметри към метод

Нека разгледаме следния код:

```
public class Numbers {  
  
    public static void main(String[] args) {  
        int n = 3;  
    }  
}
```

```

        System.out.println("Числото преди метода = " + n);
        printNumber(n);
        System.out.println("Числото след метода = " + n);
    }

    static void printNumber(int num) {
        num = 5;
        System.out.println("Числото в метода = " + num);
    }
}

```

Имаме декларирана променлива **n**, на която подаваме стойност 3. След това извикваме метод **printNumber**, на който подаваме като аргумент променливата **n**. Забележете, че в тялото на метода **printNumber** променяме стойността на параметъра, след което го отпечатваме на екрана. След излизане от тялото на метода **printNumber**, отпечатваме стойността на променливата **n**. Въпросът е – каква е стойността ѝ?

След като стартираме програмата, ще видим, че резултатът е следният:

```

Числото преди метода = 3
Числото в метода = 5
Числото след метода = 3

```



В Java предаването на аргументи към метод винаги е предаване по стойност. Това значи, че при подаване на променлива като аргумент на метод, нейната стойност се копира, когато параметъра на метода бъде инициализиран.

Дефакто при подаване на променливата **n** като аргумент на метода, ние копираме стойността 3 и инициализираме параметъра **num** с копието. Тоест в паметта вече има две променливи - **n** и **num**, и двете със стойности 3, но тези тройки са две различни и независими. Промяната на едната не се отразява на другата.

Защо е нужно да знаем всичко това ?

Както споменахме в глава 6, освен примитивни типове данни, съществуват и така наречените **референтни** типове. При тях променливите от референтен тип имат за стойност адреса на обектите в паметта. Пример за референтен тип данни са масивите.

Какво се случва, ако преправим гореописания код така, че вместо примитивна променлива, подаваме за аргумент променлива от референтен тип ? Нека видим:

```
public class Numbers {  
  
    public static void main(String[] args) {  
        int[] numbers = {2, 4, 6};  
        System.out.println("Масив преди метода = " +  
Arrays.toString(numbers));  
        printNumbers(numbers);  
        System.out.println("Масив след метода = " +  
Arrays.toString(numbers));  
    }  
  
    static void printNumbers(int[] nums) {  
        nums[0] = 5;  
        System.out.println("Масив в метода = " +  
Arrays.toString(nums));  
    }  
}
```

Резултатът от изпълнението на кода, този път, е :

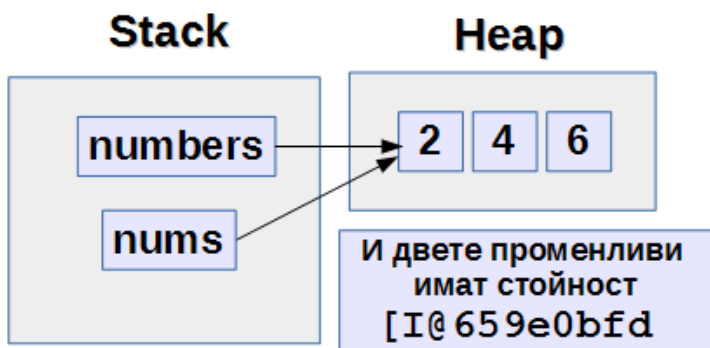
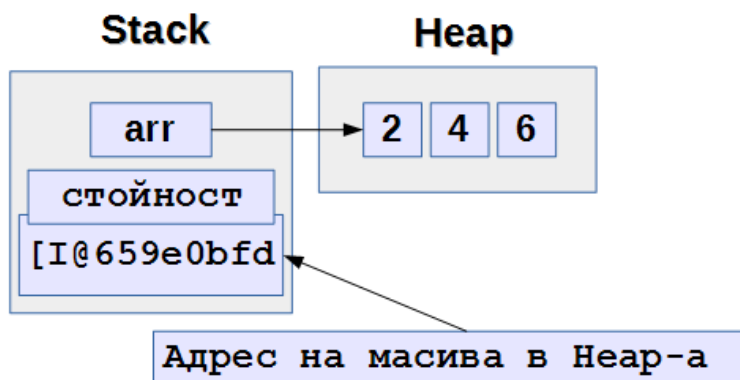
```
Масив преди метода = [2, 4, 6]  
Масив в метода = [5, 4, 6]  
Масив след метода = [5, 4, 6]
```

Нека си спомним какво представляваха променливите от референтен тип. Ако имаме следната променлива **arr**:

```
int[] arr = {2, 4, 6};
```

В паметта променливата `arr` се намира в стека, а стойността ѝ е адреса на масива в Heap-а. Дефакто стойността на променливата не е масива, а адресът към него.

Когато казваме, че в Java предаването на аргументи към метод винаги е предаване по стойност, то това важи и за референтните типове данни. Просто стойността, която се копира е адреса към истинския обект в Heap-а.



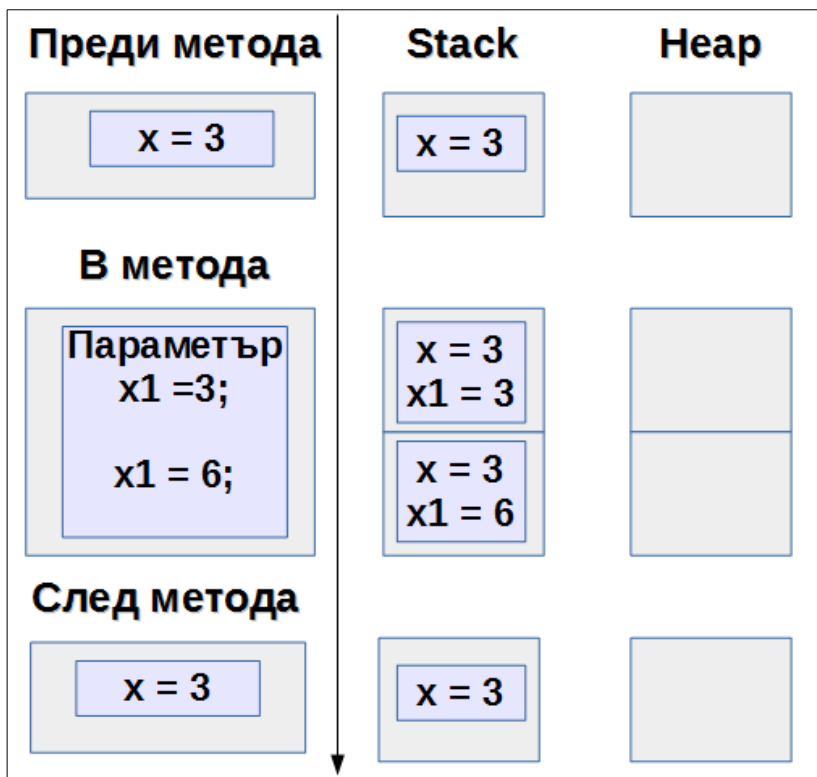
До какво водят тези факти ? Ами щом в тялото на метода модифицираме клетката на масива, към който сочи променливата **nums**, то ние променяме единствения масив в паметта, към който сочат и двете променливи. Именно затова след като излезем от метода и искаме да видим каква е стойността на масива, към който сочи променливата **numbers**, забелязваме, че стойностите на клетките са променени.



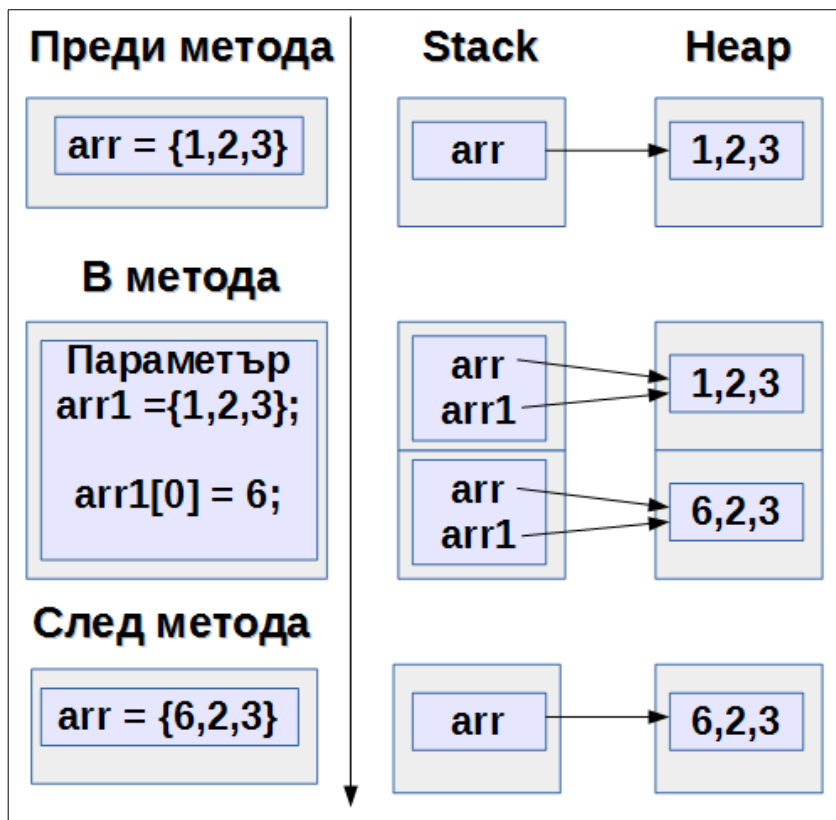
При предаване на аргументи от референтен тип към метод, се копира референцията, а не реалния обект, към който сочи тя. Промяна по копието на референцията ще се отрази на обекта и всички останали референции ще сочат към променения обект.

От особена важност е да се прави разлика, когато работим с примитивни и с референтни типове данни в методи.

Примитивните типове данни се копират и промяната на копията не се отразяват на оригиналните променливи извън цикъла.



Референтните типове данни се копират по референция – тоест в метода се създава копие на референцията, но не и на обекта, към който сочи тя. Така при промяна на променливата вътре в метода, се променя самия обект, към който сочи тя. По този начин променливите извън метода също ще сочат към променения обект.



Рекурсия

Рекурсия наричаме явлението, при което в тялото на един метод извикваме същия метод. Звучи объркващо, но в много случаи този подход прави решението на определени проблеми много просто.

Метод, който извиква себе си, се нарича рекурсивен метод.

Видове рекурсия

Рекурсията може да бъде **пряка** или **косвена**.

Пряка рекурсия имаме, когато метода извиква сам себе си.

Косвена рекурсия е такава, при която метод **А** извиква метод **Б**, а в тялото на метода **Б** се извиква метод **А**. Тази рекурсия се нарича също и неявна или непряка рекурсия.

Задължителни елементи на рекурсията

Нека дадем пример за рекурсия, за да си представим по-добре идеята:

Дадена е следната задача: Да се изпише на екрана стойността на N факториел ($N!$), където N е число, въведено от потребителя.

Факториел на едно число N представлява произведението на всички числа от 1 до N . Например $5!$ е равно на $1 \times 2 \times 3 \times 4 \times 5 = 120$.

Стъпка на рекурсията

Нека помислим над задачата: Ако вземем за стойност $N = 5$, тогава трябва да пресметнем $5!$, което е равно на $5 \times 4 \times 3 \times 2 \times 1$. Това значи, че ще умножим 5 по $(4 \times 3 \times 2 \times 1)$, което представлява $4!$. Оттук следва, че $5! = 5 \times 4!$. Същата логика следва и за $4!$ и т.н. Това означава, че за да пресметнем $N!$, ние трябва да умножим N по $(N-1)!$. По този начин виждаме, че нашият проблем съдържа в себе си операции, които са идентични – операцията по намиране на $N!$ се състои от умножение на число по резултата от същата операция, но за $N-1$. $(N-1)!$ от своя страна се изчислява чрез умножение на $N-1$ по резултата от $(N-2)!$ и т.н. Тук виждаме зависимост, която в програмирането се нарича **рекурентна връзка** или **стъпка на рекурсията**. Рекурентната връзка се реализира чрез извикването на същия метод с различни стойности на аргументите.

Дъно на рекурсията

Докога продължаваме да използваме същата операция? За $N = 5$ имаме $5! = 5 \times 4! = 5 \times (4 \times 3!) = \dots$ и така нататък, докато стигнем до използването на $1!$, което е равно на единица. Този случай, в който при определена стойност вече не е нужно да се използва същата операция, се нарича **дъно на рекурсията**.

Решението на задачата чрез рекурсия изглежда така:

```
static int factorial(int n) {  
    if(n == 1){  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```

Дъно – директен резултат

Стъпка – рекурсивно извикване на метода с различен параметър



Стъпка на рекурсията означава извикване на метода с различни аргументи от началните. Целта на стъпката е да достигне до т.н. дъно на рекурсията – до стойност на аргумента, при която изчислението ще бъде направено без ново рекурсивно извикване.



Дъно на рекурсията наричаме ситуацията, при която при определена стойност на параметрите на метода няма да се направи нова стъпка, а директно ще се върне резултат.

Без дъно и стъпка не можем да осъществим работеща рекурсия.

Безкрайна рекурсия

Рекурсия без дъно или рекурсия, чиято стъпка не променя стойността на параметрите, наричаме **безкрайна рекурсия**. Безкрайната рекурсия е подобна на безкрайния цикъл, но ефекта над приложението е различен. Докато при безкрайния цикъл програмата ще се изпълнява вечно, то при безкрайната рекурсия ще се извиква безкраен брой пъти подадения метод и никога няма да се върне резултат. Това ще доведе до запълване на паметта на приложението, след което програмата ще прекрати изпълнение.

Добри практики при рекурсията

Използването на рекурсия може в някои случаи да ни доведе до много елегантни и лесно четими решения. Трябва да се знае, обаче, че ако не се използва правилно, рекурсията може да повлияе отрицателно на бързината на приложението.

За да използваме рекурсия, винаги трябва да подходим към задачата с ясната цел да разбием проблема на подпроблеми от същия тип. Вземете за пример задачата за факториел-а. За да открием рекурентната зависимост, избрахме едно произволно число (в нашия случай числото 5) и се опитахме да резлизирате неговия факториел, като разпишем формулата. По този начин видяхме, че операцията по смятането на $5!$ се състои от умножение на 5 с операцията по смятането на $4!$, което означава, че проблема за изчислението на $N!$ Представява $N \times (N-1)!$

Откриването на рекурентна връзка не винаги е лесно. Често е нужно да разпишем задачата с примерни стойности, за да открием зависимост.

Рекурсия и итерация

Всеки рекурсивен алгоритъм може да се реализира чрез цикли. За да използваме рекурсивен подход при решаването на даден проблем, трябва да сме убедени, че решението ще е елегантно и лесно четимо. Неправилното използване на рекурсията може да доведе до бавен и много трудно поддържан код.

Упражнения

4. Да се състави метод, който приема като аргументи две числа и отпечатва на екрана техния сбор.
5. Да се състави метод, който приема като аргументи две числа и връща тяхното средноаритметично.
6. Да се състави метод, който приема като аргументи два масива и връща масив, съдържащ всички елементи на двата подадени масива.

7. Да се състави метод, който приема като аргументи две числа X и Y и връща като резултат двумерен масив с X реда и Y колони, като във всяка клетка на масива се намира цифрата 0.
8. Да се състави програма, която изчислява N -тото число на Фибоначи. Първото и второто число на Фибоначи са 1-ци, всяко следващо е равно на сбора на предходните 2.

Пример: 8

Изход: 21

Първите 8 числа на Фибоначи са: 1, 1, 2, 3, 5, 8, 13, 21.

Глава 9. Символни низове

В тази глава

Въведение в символните низове.

Декларация, инициализация, вход и изход от конзолата.

Основни операции.

Представяне на символните низове в паметта.

Допълнителни операции върху символни низове.

Класът `StringBuilder` – основни операции и употреба.

Въведение в низовете

Много често ни се налага да работим с текст, по-дълъг от един или няколко символа. Използвайки знанията си до момента, можем да направим масив от знаци, в който да четем и да обработваме текстови данни. Недостатък на това решение е, че много често ни се налага да ползваме основни операции с текст – търсене на текст в друг, вмъкване, изтриване, форматиране и други. Друг проблем е, че масивът от знаци е с фиксирана дължина. Това много често не ни върши работа, защото не винаги знаем предварително дължината на текста, който ще прочетем. В Java съществува вграден тип данни, който ни позволява да работим с текстови данни. Това е типът **String** и се използва за променливи, които съхраняват текст под формата на символен низ.

Декларация, инициализация, вход и изход от конзолата

Текстови низове се създават, като се създаде променлива от тип String и се инициализира с текстов литерал, ограден с двойни кавички :

```
String someText = "Примерен текст записан в променлива от тип String";
```

Забележете, че типът String не е запазена дума в Java, както досега изучените примитивни типове данни. Това е така, защото String не е примитивен тип. Въпреки това, String може свободно да се ползва във всяка наша програма без изрично да указваме каквото и да било (спомнете си как указвахме, че ще използваме Scanner).

Извеждането на конзолата на променлива от тип String е лесно :

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

За да прочетем низ от конзолата използваме отново класа Scanner и операцията nextLine(), която ще ни прочете един цял ред и ще ни го върне.

```
Scanner sc = new Scanner(System.in);  
String lineOfText = sc.nextLine();
```

Когато въвеждаме текстов низ по този начин, след всяко натискане на клавиша **Enter** конструкцията **next()** ще ни прочита по един ред и ще ни го връща като резултат. В горния пример ние прочитаме само един ред и го записваме в променливата **lineOfText**.

Основни операции с текстови низове

String ни дава голям брой операции за работа с текстови данни, като в настоящата глава ще разгледаме само най-основните от тях.

Важно е да отбележим, че променливите, които създаваме от тип **String** са непроменими (**immutable**). Понеже **String** не е примитивен тип, то стойностите на променливи от тип **String** се съхраняват в Heap-а. При **immutable** променливи операциите, които модифицират по някакъв начин стойността, водят до създаване на нова стойност в Heap-а, като старата вече не се използва. Ще разгледаме по-нататък в настоящата глава как се представят низовете в паметта.

Дължина на низ

Можем да вземем дължината на низ използвайки метода **length()**:

```
Scanner sc = new Scanner(System.in);
String text = sc.nextLine();
System.out.println("Дължината на текста е:" + text.length());
```

Достъпване и обхождане символите на даден низ

Знаците в един низ са също номерирани (индексирани), както в масива. Номерацията започва от нула и стига до дължината на низа минус едно. Можем да прочетем знака на дадена позиция в низ, като използваме метода **charAt()**.

```
String text = "Здравей, аз съм текст. Идвам с мир.";
System.out.println("Петият знак е:" + text.charAt(4));
```

Можейки да вземем знак на определена позиция, броя знаци и знаейки как са номерирани, може да обходим всички елементи на един текстов низ използвайки цикъл **for** по всички валидни индекси и след това с

метода `charAt` да вземем поредния символ. Имайте на предвид, че по този начин можем само да прочетем даден символ, но не и да го променим.

```
String text = "Здравей, аз съм текст. Идвам с мир.";
for(int index=0; index<text.length(); index++) {
    //извеждаме всички знаци на отделни редове
    System.out.println(text.charAt(index));
}
```

Пример: брой думи в даден текст

Като пример за обхождането на всички символи в низ ще решим следната задача: по даден текст да преброим броя думи. Ще считаме, че думите са последователност от символи, разделени с интервал, без да броим препинателни знаци, цифри и други символи за разделители между думите. В този си вид задачата се свежда до това да преброим интервалите, които разделят думите. Ще започнем броенето от едно, защото преди първата дума няма интервал, затова предварително я включваме в бройката. След това ще обиколим всички символи и ще проверим всеки символ са равенство със знака за интервал, представен като променлива от тип знак (`char`).

```
Scanner sc = new Scanner(System.in);
// прочитаме текста от конзолата
String text = sc.nextLine();

// започваме от 1, защото имаме поне 1 дума
int wordCount = 1;
// за всички символи
for (int index=0;index<text.length();index++) {
    // ако открием интервал, то това е начало за нова дума
    // и преброяваме
    if (text.charAt(index) == ' ') {
        wordCount++;
    }
}
System.out.println("Броят думи е " + wordCount);
```


Слепване (конкатенация) на низове

Можем да получим низ като резултат от долепването на два други низа един с друг. За тази цел използваме метода `concat()`, който приема като параметър друг низ и връща нов, трети низ – резултат от слепването на първите два.

```
String firstName = "Иван ";
String lastName = "Петров";
System.out.println(firstName.concat(lastName));
```

За постигане на същата цел по-често се използва оператора "+", който прави абсолютно същото и резултатът отново е нов String, който съдържа двата низа долепени един до друг.

```
String firstName = "Иван ";
String lastName = "Петров";
System.out.println(firstName+lastName);
```

Операторът "+" може да се използва, за да долепим променливи от всякакъв тип към даден низ. Освен това, тъй като резултатът е нов низ, към него също можем да продължим да долепваме данни. Това ни помага да конструираме по-сложни изрази.

```
String firstName = "Иван";
String lastName = "Петров";
int age = 25;
//конструираме нов низ чрез слепване на две променливи от тип
String, едно цяло число и три низови литерала
String result = firstName + " " + lastName + " е на възраст "
+ age + " години.";
System.out.println(result);
```



Използвайте скоби, когато имате комбинация от аритметични изрази и слепвания с низове. В противен случай може да получите неочаквани резултати. Ето за пример следния код :

```
int a = 3;
int b = 5;
```

```
System.out.println("Резултатът е : " + a + b);
//Резултатът е : 35
```

Тъй като изразът започва с низов литерал, то от там нататък операторът "+" действа като конкатенация(долепване), а не като аритметичния оператор "+". Можем лесно да поправим това като използваме скоби :

```
int a = 3;
int b = 5;
System.out.println("Резултатът е : " + (a + b));
//Резултатът е : 8
```

Представяне на низовете в паметта

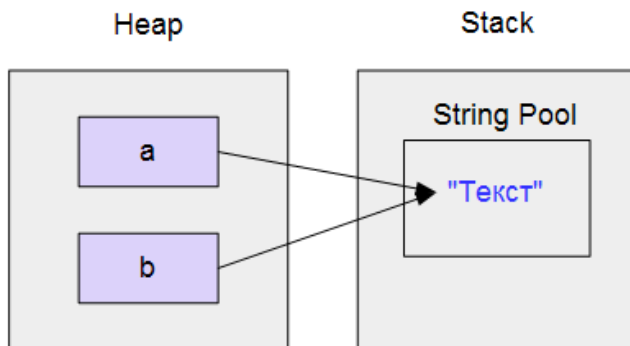
За да се оптимизира използваната памет, низовете в Java се съхраняват по по-различен начин от другите обекти и примитивни променливи. Ще видим кое е по-специфичното и защо се налага низовете да са непроменими (**immutable**) обекти.

Пул(pool) от обекти

За да се пести памет, низовите литерали се съхраняват в отделен отсег от паметта, които се преизползват. Това, което се случва, е че те се съхраняват в определено място в паметта, наречено пул от низове (**String pool**). Когато използваме даден низ, се проверява дали той вече не съществува в пула от низове в паметта. Ако го няма – се създава нов **String**, в противен случай променливата запазва адреса към вече съществуващия такъв в тази памет. Ще илюстрираме това със следния пример: нека да имаме два низа "a" и "b". Нека да им присвоим еднакъв символен низ, за да видим как ще изглеждат те в паметта.

```
String a = "Текст";
String b = "Текст";
```

В този момент паметта изглежда по следния начин :



Лесно можем да проверим, че двете референции сочат към една и съща памет. Както знаем, когато сравняваме променливи от референтни типове данни, каквито са низовете, то ние проверяваме дали те сочат към едно и също място в паметта. Така че оператора "==" ще ни покаже дали двете променливи сочат една и съща памет.

```
String a = "Текст";
String b = "Текст";
System.out.println(a==b); // извежда true
```

Сравнение на низове

Следвайки горния пример, може да предположим, че низове се сравняват с оператора "==". Това не винаги е вярно и затова има друг начин за сравнение, който ще разгледаме по-късно. За да разберем кога това няма да ни даде правилен резултат, ще разгледаме още един начин за създаване на String – използвайки оператора "new" :

```
String myString = new String("Примерен текст");
```

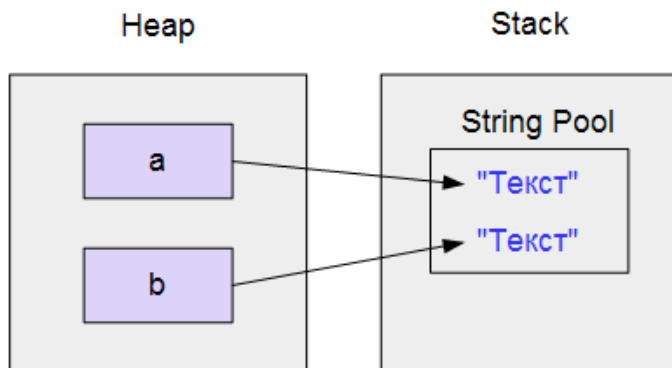
Оператора "new" се използва за конструиране String обекти и по други начини, например по даден масив от знаци :

```
char[] chars = {'з', 'д', 'п', 'а', 'в', 'е', 'й'};
String myString = new String(chars);
```

Какво се случва в паметта в този момент? Операторът "new" създава нов символен низ, без да се проверява дали неговата стойност вече съществува в пула от низове. С други думи, ако променим примера от по-горе така, че низовете "a" и "b" да се създават по следния начин:

```
String a = new String("Текст");
String b = new String("Текст");
```

то в паметта обектите и референциите към тях ще бъдат следните:



Ясно се вижда, че вече имаме два различни обекта и сравнението с оператора "=", няма да даде правилния резултат, без значение че двата низа като текстови данни са едни и същи :

```
System.out.println(a==b); // резултата ще бъде false
```

От тук следва, че низовете, както и масивите, не се сравняват с оператора за двойно равенство. Правилният начин е като използваме метода equals() :

```
String a = new String("Текст");
String b = new String("Текст");
System.out.println(a.equals(b)); // вече резултата ще бъде true
```

Методът "equals()" проверява дали съдържанието на два текстови низа съвпада. Този път няма значение накъде в паметта сочат двете референции "a" и "b".

Други начини за сравнение на низове

equalsIgnoreCase()

Можем да сравним два низа без да има значение от големи и малки букви, използвайки `equalsIgnoreCase()`:

```
String a = new String("Текст");
String b = new String("текст");
System.out.println(a.equalsIgnoreCase(b)); //резултата ще бъде
true независимо от разликата в големи/малки букви
```

compareTo()

Методът `compareTo()` сравнява два низа, като ни връща кой от тях е по-големия, използвайки лексикографска подредба. Най-просто казано това е начинът, по който са подредени символните низове в телефонния указател – т.е. първо са числата, след това големите и след това малките букви, подредени по естествения им ред. Методът връща число от тип `int`, като са изпълнени следните условия:

- ако низът, върху който извикваме метода, е по-малкият, то резултатът е число по-малко от нула.
- ако двата низа са еднакви, то резултатът е точно нула.
- ако низът, върху който извикваме метода, е по-големият, то резултатът е число по-голямо от нула.

Можем да разгледаме използването на метода в следните примери:

```
String string1 = "Петър плет плете.";
String string2 = "Кон боб яде ли?";
String string3 = "Да програмираш или да простираш - това е
въпросът?";

// положителен резултат, защото първият стринг е
лексикографски по-голям от втория
int result1 = string1.compareTo(string2);
System.out.println(result1);

// негативен резултат, защото третият стринг е лексикографски
по-малък от втория
int result2 = string3.compareTo(string2);
```

```
System.out.println(result2);

// нула, защото първият стринг съвпада с първия
int result3 = string1.compareTo(string1);
System.out.println(result3);
```

Допълнителни операции върху низове

split()

Използваме метода **split()**, за да разделим даден текст по даден разделител. Като резултат ще получим масив от String стойности, които представляват частите на оригиналния текст, разделени от разделителя. Примерно по даден текст от думи, разделени със запетая, ще получим масив от всички думи. Ето пример за това :

```
// имена, разделени със запетая, в един текстов низ
String text = "Пешо,Гошо,Мишо,Стефка,Иванка";

// разделяме по запетая и получаваме масив от низове с всички
// имена като отделни елементи в масива
String[] names = text.split(",");

//обикаляме всички имена в масива и ги извеждаме
for (String name : names) {
    System.out.println(name);
}
```

join()

Методът **join()** работи по точно обратния начин на метода **split()** - по даден стринг – разделител и масив от низове, получаваме низ, резултат от слепването на низовете в масива, разделени с подадения разделител. Примерно по дадени думи можем да получим текст, като думите ще са разделени с точка и запетая. Ето пример за това :

```
// имена разделени със запетая в масив от низове
String[] names = {"Пешо", "Гошо", "Мишо", "Стефка", "Иванка"};
// съединяваме низове като ги разделяме с ";"
String textWithNames = String.join(";", names);
```

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

substring()

Използваме метода `substring()`, за да извлечем някаква част(подниз) от даден низ. Методът приема един аргумент – началният индекс, от който да започне вземането на подниза. Имаме и втори вариант, който приема и крайната позиция. Важното тук е да отбележим, че извлечения подниз съдържа знаците на позиции от началото до края минус едно. Ето пример за това :

```
String text = "Днес е хубав ден за работа със низове.";
System.out.println(text.substring(5)); // извежда от 6-тия
знак до края (от този с индекс 5)
System.out.println(text.substring(0,4)); // извежда от първия
до 4-тия знак (до този с индекс 3)
```

Изходът от този пример е следния :

```
е хубав ден за работа със низове.
Днес
```

replace()

Извикваме метода `replace()` върху даден низ, за да заменим всички срещания на даден подниз - с друг. Методът приема два аргумента – какво търсим и с какво да го заменим. Имаме и втори вариант, който приема две променливи от знаков тип(`char`), като така методът заменя срещанията на първия символ с втория. Не трябва да забравяме, че низовете са **immutable**, така че резултатът от тази операция е нов низ, докато старият ще остане непроменен. Можем да видим това в следния пример :

```
String originalText = "C# е страхотен език за програмиране и
на C# се пише лесно";
// заменя всяко срещане на текста 'C#' с текста 'Java'
// забележете, че резултатът се връща и е записан в нова
```

```
променлива, а 'originalText' ще остане непроменена
String correctedText = originalText.replace("C#", "Java");
System.out.println("Коригиран текст : " + correctedText);
System.out.println("Оригинален текст : " + originalText);
```

trim()

Използваме метода trim(), за да премахнем интервалите в началото и края на даден низ. Празните пространства, оградени от други знаци, няма да бъдат премахнати. Методът не приема аргументи. Важно е отново да отбележим, методът не модифицира оригиналния низ, а връща нов като резултат. Ето пример за това :

```
String originalText = "    Текст с празни пространства в краищата.    ";
// премахваме интервалите в краищата и записваме резултата в нова променлива
String trimmedText = originalText.trim();
System.out.println(trimmedText);
```

indexOf()

Използваме метода indexOf(), за да намерим първата позиция, на която даден низ се среща в друг. Методът приема един параметър – низа или знака, който ще търсим. Имаме и втора форма, която приема начална позиция от която да започнем търсенето. Нека да разгледаме следните примери:

```
String text = "Програмирането е изкуството на мисълта.";
// намираме къде в текста се намира думата "изкуство"
// резултата е от тип цяло число
int artPosition = text.indexOf("изкуство");
System.out.println(artPosition);
// намира позицията на знака 'e' в текста
int isPosition = text.indexOf('e');
System.out.println(isPosition);
```


Методите toLowerCase() и toUpperCase()

Двойката методи `toLowerCase()` и `toUpperCase()` се използват, за да получим нов низ, като всички букви са обърнати само в малки (`toLowerCase`) или само в главни (`toUpperCase`). Не трябва да забравяме, че оригиналният низ остава непроменен, а резултата ни се връща и ние трябва да си го запишем в нова или в същата променлива.

```
String text = "Примерен текст с Малки И ГолемИ БукВи.";
// преобразуваме текста, като сменяме големите букви с малки
System.out.println(text.toLowerCase());
// преобразуваме текста, като сменяме малките букви с големи
System.out.println(text.toUpperCase());
```

Резултатът от примера е следния :

```
примерен текст с малки и големи букви.
ПРИМЕРЕН ТЕКСТ С МАЛКИ И ГОЛЕМИ БУКВИ.
```

Форматиране на низове с метода String.format()

Извикваме метода `String.format()`, за да конструираме нов символен низ по зададен шаблон. Шаблонът дефинираме, като запишем символен низ, съдържащи специални символи за „параметри“, както и списък от променливи, които да бъдат поставени в символния низ на местата на обозначените параметри. Методът приема шаблон – символен низ и след това толкова на брой променливи, колкото участват в шаблона. За да обозначим, че на дадена позиция в шаблона ще се вмъкне стойност на променлива, използваме знака процент (%), следван от конкретна форматираща комбинация. Нека да видим пример за това:

```
int age = 29;
String name = "Гошо";
float money = 5.1523f;
// създава низ, който вмъква променливите на подходящите
// места, форматирани по съответния начин
String text = String.format("%s има %.2f лева и е на %03d
години.", name, money, age);
System.out.println(text);
```

Изходът е следният:

Гошо има 5,15 лева и е на 029 години.

Важно е отбележим, че на местата в шаблона, отбелязани с %, се поставят стойностите на променливите в същия ред, в който са зададени. Какво означават останалите символи от този шаблон? Нека да разгледаме общия вид първо:

"% [флаг] [дължина] [.точност] тип"

Полетата заградени с "[]" не са задължени да бъдат отбелязани. Полетата носят следната форматираща информация:

- **флаг** – показва как да бъдат форматирани и подравнени числа и низове. Стойността "+" означава, че пред числа винаги ще има знак "+". Стойност "0" означава, че числото ще бъде допълвано с нули до някаква зададена дължина. Ако зададем низ вместо число, знаците "+" и "-" показват дали низът ще бъде допълнен с интервали отляво или отдясно.
- **дължина** – показва минималната дължина, която трябва да заема стойността, която ще отпечатаме. Ако дължината на числото или низа е по-малка, тя се допълва с нули при числата или с интервали при низовете.
- **точност** – задава до колко точно знака след десетичната запетая да се изведе стойност. Това важи за числата с плаваща запетая.
- **тип** – показва типа на променливата, която ще изведем. Възможните стойности са : "d" – за цели числа, "f" – за числа с плаваща запетая, "s" – за низове и "x" – за да ни излезе зададено цяло число в шестнадесетичен формат.

contains()

Използваме метода contains(), за да проверим дали даден низ се съдържа в друг. Методът приема един параметър – низа, който ще търсим. Резултатът е от тип boolean – true, когато се съдържа и false – в противен случай. Важно е да отбележим, че contains прави разлика между малки и големи букви, когато проверява дали търсеният низ се среща. Можем да видим това в следния пример :

```
String text = "Примерен текст.";
boolean hasText = text.contains("текст");
boolean hasText2 = text.contains("Текст");
System.out.println("Наличие на текст : " + hasText);
System.out.println("Наличие на Текст : " + hasText2);
```

Класът StringBuilder – основни операции и употреба

Когато ни се налага често да променяме даден низ, класът String, не е особено подходящ. Това е така поради факта, че всички операции върху обект от тип String създават и връщат нов обект, вместо да променят съществуващия. Това може да бъде доста бавен и отнемаш много памет процес, както ще видим към края на настоящата част. Когато ни се наложи много пъти да модифицираме един символен низ, е удачно да ползваме друга конструкция, при която модификацията не създава винаги нова стойност в паметта. По този начин пестим както памет, така и време. Конструкцията се нарича **StringBuilder**.

StringBuilder е тип данни, аналогичен на String – символен низ. Разликата е в това, че промяната на стойността на StringBuilder променлива не води до създаването на нова стойност в паметта. Затова казваме, че **StringBuilder е mutable String**.

Деклариране и инициализация

StringBuilder можем да създадем по няколко начина. Можем да StringBuilder без никаква стойност, като след това можем да допълним вътре данни. Друг вариант е да създадем StringBuilder, като използваме вече съществуваща String стойност. Нека да разгледаме няколко примера :

```
// празен StringBuilder
StringBuilder myBuilder = new StringBuilder();
// StringBuilder, инициализиран със стрингов литерал
StringBuilder builder = new StringBuilder("Текст в
StringBuilder");
// StringBuilder инициализиран със обект от тип String
String text = "Пример";
StringBuilder builder2 = new StringBuilder(text);
```

Модификация на `StringBuilder` обекти – методите `insert`, `delete`, `replace` и `append`

`insert()`

Използваме метода `insert()`, за да вмъкнем даден низ, число, булева стойност и т.н. на специфична позиция в нашия символен низ. Методът приема два параметъра – позиция и стойност, която да се вмъкне на тази позиция. Номерацията, както и при низовете, започва от нула. Важно е да отбележим, че методът директно променя символния низ, върху който се изпълнява. Можем да видим това в следния пример:

```
StringBuilder builder = new StringBuilder("текст в  
StringBuilder ");  
// вмъкваме низ в началото  
builder.insert(0, "Примерен ");  
// вмъкваме число в края  
builder.insert(builder.length(), 1);  
// отпечатваме, както печатаме String променливите  
System.out.println(builder);
```

`delete()`

Използваме метода `delete()`, за да изтрием част от съдържанието на даден `StringBuilder`. Методът приема два параметъра – начална и крайна позиция, от която да се изтрие текст. Нека да разгледаме следния пример:

```
StringBuilder builder = new StringBuilder("Примерен текст в  
StringBuilder");  
// изтриваме " ен текст"  
builder.delete(6, 14);  
System.out.println(builder);
```

`replace()`

Използваме метода `replace()`, за да заменим част от нашия текст с друг текст. Методът приема три параметъра – две позиции и стойност, която

да замени текста на тези позиции. Можем да видим това в следния пример :

```
StringBuilder builder = new StringBuilder("Примерен текст в  
StringBuilder");  
// заменяме " ен текст" с низа от по-долу  
builder.replace(6, 14, " като текст, съхранен");  
System.out.println(builder);
```

Резултатът от горния пример е следния:

Пример като текст, съхранен в StringBuilder

append()

Използваме метода `append()`, за да вмъкнем съдържание в края на някакъв текст. Методът приема един параметър – стойността, която да се долепи към края на текста. Можем да видим това в следния пример :

```
StringBuilder builder = new StringBuilder("Примерен текст");  
//долепяме в края  
builder.append(" и още текст.");  
System.out.println(builder);
```

Сравняване на стойности от тип StringBuilder

Сравняването на съдържанието на стойности от тип `StringBuilder` не става нито с оператора за сравнение, нито с операцията `equals()`. Един от най-правилните начини за това е да преобразуваме `StringBuilder` стойността в такава от тип `String`, използвайки операцията `toString()` и след това да сравняваме, използвайки методите за сравнение на низове, които вече разгледахме в настоящата глава. Нека да видим как става това:

```
StringBuilder builder = new StringBuilder("Примерен текст");  
StringBuilder builder2 = new StringBuilder("Примерен текст");  
// преобразуваме в низове  
String string1 = builder.toString();  
String string2 = builder2.toString();  
// сравняваме низовете  
System.out.println("Равни ли са:" + string1.equals(string2));
```

Сравнение с класа String

| String | StringBuilder |
|--|---|
| Обектите от клас String са непроменими (immutable). | Обектите от клас StringBuilder са променими (mutable). |
| Слепването на много низове изисква много време и памет, защото всеки път се създава нов обект. | Слепването на много низове, както и на други данни към даден StringBuilder обект е бързо и не отнема много памет, защото се модифицира една и съща стойност. |
| Стойности от тип String се сравняват с операциите equals , equalsIgnoreCase и compareTo . | Стойности от тип StringBuilder трябва да се обърнат до тип String с операцията toString() и след това да се сравняват. |

Упражнения

1. Да се състави програма, чрез която от клавиатурата се въвеждат последователно две думи с дължина до 20 знака. Програмата да размени първите им 5 знака и да изведе дължината на по-дългата, както и новото ѝ съдържание.

Пример:

uchilishe uchenik

Изход:

9 uchenishe

2. Да се състави програма, чрез която се въвеждат последователно две редици от символи без интервали. Програмата да извежда съобщение за резултата от сравнението им по позиции.

Пример:

хипопотам

хипопотук

Изход:

Двата низа са с равна дължина.

Разлика по позиции:

8 а-у

9 м-к

3. Да се състави програма, чрез която се въвеждат 2 редици от знаци. Ако в двете редици участва един и същи знак, да се изведе на екрана първата редица хоризонтално, а втората вертикално, така че да се пресичат в общия си знак. Ако редиците нямат общ знак да се изведе само уведомително съобщение.

Пример :

машина

шапка

Изход:

м

а

шапка

и

н

а

Упътвания

- Проверете дължината на думите използвайки метода `length()`. Обиколете по-късата дума с цикъл като залепвате в нов низ или в `StringBuilder` първите 5 знака от тази дума, а след това и останалите знаци от по-дългата дума.
- Сравнете дължините, изкарайте на базата на това подходящо съобщение и завъртете цикъл до по-малката дължина. В цикъла

сравнявайте символ от единия низ със символ от другия за всяка валидна позиция на символ.

➤ Проверете за всеки символ на първия низ къде се среща във втория използвайки `indexOf()`. След това изкарайте първия низ символ по символ, но преди изведете необходимия брой интервали, отново с цикъл. На позицията, на която имат общ символ – изведете втория низ.

Глава 10. Въведение в алгоритмите

В тази глава

Какво е алгоритъм

Защо е нужно да изучаваме алгоритми

Сложност на алгоритми – опростено обяснение

Какво е алгоритъм

Алгоритъм наричаме съвкупността от операции, поредени в определен ред, посредством които се решава даден проблем.

Досега, при разглеждането на примерни задачи в тази книга, винаги сме описвали стъпките, които трябва да предприемем, за да решим проблема или да изпълним условието. Именно описването на стъпките и подреждането им в определен ред се нарича описване на алгоритъм на задачата.

За разрешаването на определен проблем могат да съществуват много алгоритми, тоест много начини, по които да се подходи.

В живота и в работата си използваме алгоритми постоянно, без да осъзнаваме. Например за да стигнем сутрин от вкъщи до университета, на нас ни е нужен алгоритъм – последователност от стъпки, които трябва да изпълним:

- да излезем от вкъщи;
- да се придвижим с транспорт до адреса на университета;
- да влезем в университета.

Стъпките трябва да са подредени точно по един начин – не можем , например, да влезем в университета преди да сме излезли от вкъщи.

Можем да използваме различен алгоритъм от описания, за да извършим нужното в задачата – примерно можем да отидем пеша до университета или да се отбием с колата до мола, преди да отидем до университета. Тоест съществуват различни алгоритми за решаването на един проблем.

Защо е нужно да изучаваме алгоритми

Програмирането в основата си се състои от постоянно решаване на проблеми. Всяка една задача, която е поставена пред вас, е проблем, който трябва да се реши. За да станем добри програмисти, преди всичко трябва да се научим да решаваме проблеми. Добре, но как ? Ами като се измисли алгоритъм за тях. Как решаваме проблема „отиди до университета“ ? Измисляме последователност от стъпки, които, събрани заедно, довеждат до решението. Аналогично всяка задача, която ни се даде, може да се реши с краен брой последователни стъпки. Ако измислим стъпките, след това лесно можем да ги облечем в код и да напишем решението на задачата.

Именно затова изучаваме алгоритми и те са застъпени до голяма част в настоящата книга. Целта е да добием представа за начина на мислене и

подхода, когато се изправим пред проблем. Също така изучаването на основни алгоритми ни помага да се запознаем с някои стандартни практики за решаване на малки тривиални проблеми. По този начин можем да преизползваме тези методи, за да сглобяваме алгоритми за по-сложни задачи, когато имаме нужда.

Около 85% от решаването на една задача е измислянето на алгоритъм, по който да се придържаме за изпълнението ѝ. При готов алгоритъм единственото, което ни остава да направим, е всяка стъпка да се сведе до инструкция от код, посредством които програмата ни придобива цялостен и завършен вид. Тоест писането на код е едва 15% от работата. Ето защо е от изключителна важност да имаме алгоритмично мислене, тоест да можем да измисляме алгоритми за решаването на проблеми.

Откъде знаем как да стигнем до университета ако ни се наложи? Как сме се сетили, че трябва да вземем транспорт или да вървим по пътя? Откъде знаем как се влиза в университета? Отговорите на тези въпроси са „защото сме го правили преди“ или „защото сме виждали как се прави“ или „защото първия път ме заведоха“. С две думи отговорът е „имам опит“. По един или друг начин на нас ни е известно съществуването на определени начини за решаване на тривиални проблеми – как се влиза в сграда, как се върви по улицата и т.н. Тези решения ги взимаме за даденост, когато решаваме по-сложния проблем „как да отида до университета“. Тези дребни стъпки, колкото и малки и прости да изглеждат, също представляват своеобразни алгоритми – начини, по които се решават проблеми. Но тези стъпки са ни ясни, защото сме се сблъскали с тях преди.

Изучаването на алгоритми ни позволява да се запознаем с такива малки стъпки, тривиални проблеми, както и със стандартните методи за решаването им. По този начин ние добиваме така ценния опит, който после ни помага сами да сглобяваме сложни алгоритми и да решаваме по-мощни проблеми.

Как да измислим алгоритъм?

Нека вземем самата дефиниция на алгоритъм, а именно – съвкупност от операции, поредени в определен ред. Това означава, че за да изградим един алгоритъм, просто трябва да съставим списък с по-малки стъпки, които са нужни, за да решим нашия проблем. Всяка стъпка може

да представлява независим подпроблем, чието решение се състои отново от алгоритъм.

Например нека вземем проблема „трябва да отидем до университета“.

Стъпките, които трябва да предприемем са:

- да излезем от вкъщи;
- да се придвижим с транспорт до адреса на университета;
- да влезем в университета.

Стъпка 2 сама по себе си е сложен проблем, за който също трябва да измислим решение. Тоест стъпка 2 има нужда от алгоритъм. Какво е нужно, за да се придвижим с транспорт до адреса на университета ?

2.1. Да стигна от вкъщи до спирка на автобус;

2.2. Да се кача на автобуса;

2.3. Да слеза на правилната спирка;

2.4. Да се придвижа от спирката до адреса на университета

Всяка една от тези стъпки може също да се състои от списък с подзадачи и така нататък, докато алгоритъма не се разбие до крайно множество от подробни стъпки, нужни да се изпълнят.

Всеки алгоритъм има т.н. **най-малка съставна част**. Това представлява стъпка, чието решение е тривиално.

Ако ние сведем нашият алгоритъм до последователност от краен брой най-малки съставни части, то тогава нашият сложен проблем ще представлява съвкупност от много малки тривиални проблеми, чиито решения са лесни и ясни.

Едва тогава можем спокойно да кажем, че сме измислили решение на задачата и това, което остава да направим, е да напишем кода, който да изпълни нужните инструкции.

Нека сега направим друг пример, този път със задача по програмиране: да се намери най-голямото число от масив с числа. Какъв е алгоритъма за решение на задачата ?

10. Създаваме си една променлива, в която запазваме първият елемент на масива;

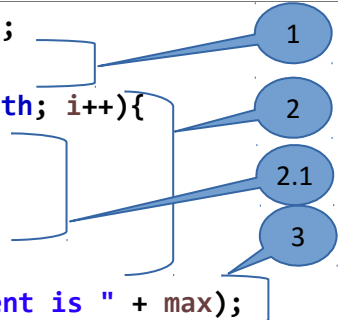
11. Обхождаме всички останали елементи, като проверяваме дали не сме срещнали по-голям от вече запазения.

1. Ако да, то в допълнителната променлива записваме стойността на по-големия елемент.

12. Така след като приключим обхождането, в допълнителната променлива ще се съдържа най-големия елемент в масива.

Кодът, който трябва да реализираме, след като знаем всички стъпки, е следния:

```
int[] array = {5,2,6,8,1,4,2};  
int max = array[0];  
for(int i = 0; i < array.length; i++){  
    if(array[i] > max){  
        max = array[i];  
    }  
}  
System.out.println("max element is " + max);
```



Забележете, че всяка стъпка се трансформира до елементарна инструкция от код – цикъл, условен оператор или обикновена инструкция за изписване на данни. Затова е важно нашите алгоритми да са разбити до най-малките им съставни части. Едва тогава можем да сме сигурни, че всяка част на алгоритъма ще може да се реализира лесно и просто.

Чрез измислянето на алгоритъм на една задача, ние превръщаме големи сложни проблеми в съвкупности от малки и прости проблеми.

Сложност на алгоритми – опростено обяснение

Да се върнем на примера с ходенето от вкъщи до университета. Можем да стигнем до университета чрез кола, градски транспорт, колело, пеша и т.н. Можем да отидем директно от вкъщи към университета, можем да минем през центъра, по околоръстното или през село. Кой подход, обаче, е най-бърз ? А кой е най-евтин ? Дали да пътуваме с кола до университета ? Дали пеша ? Дали да не се отбием до морето преди да стигнем до университета ? И трите варианта ще ни отведат до университета. Но колко бързо ? А на каква цена ?

Алгоритмите за решаването на един проблем се различават главно по две неща – бързина и цена.

Така е и в програмирането – бързината на един алгоритъм зависи от неговата **сложност по време**, а ресурсите, които са нужни за изпълнението му зависят от неговата **сложност по памет**.

Сложност по време

Сложността по време на един алгоритъм представлява степента на изменение в скоростта на алгоритъма спрямо изменението на обема от входни данни.

Нека вземем за пример задачата за намиране на максималния елемент в масив, която решихме преди малко. За масива въведохме 7 стойности - 5,2,6,8,1,4 и 2. Ще се промени ли кодът, ако вместо 7 цифри, в масива сложим 70 ? А 700 ? Кодът няма да се промени. Но времето, за което ще се изпълни алгоритъма ще бъде различно. Това е така, защото цикълът, който проверява за максимален елемент, ще се завърти толкова пъти, колкото елементи имаме в масива. Следователно условната операция за проверка на следващия елемент ще се изпълни толкова пъти, колкото елементи имаме.

Сложност по памет

Сложността по памет на един алгоритъм представлява степента на изменение в заетата памет от алгоритъма спрямо изменението на обема от входни данни.

По време на изпълнението на алгоритъма, ние заделяме допълнителни променливи, които ни помагат да решим задачата. В случая с нашата задача, ние заделяме променливата „max“ и работим с нея по време на изпълнението на алгоритъма. Тази променлива заема определен обем памет в компютъра. Съответно колкото повече променливи и структури от данни използваме, толкова повече памет се заделя за алгоритъма. Ако броя на клетките в масива се утрои, обема памет, който ще запълним, за да решим задачата няма да се промени, защото нашата променлива „max“ ще ни свърши работа за произволен размер на масива. Следователно паметта, която ще е нужна, за изпълнението на алгоритъма, няма да се промени, дори при промяна на размера на входните данни.

Видове сложност на алгоритми

Най-общо казано сложността на алгоритмите отразява изменението на времето или паметта, нужна за изпълнение на алгоритъма, спрямо изменението на входните данни.

Сложността на алгоритмите е основният начин за сравняване на два алгоритъма. Сложността по време ни дава информация кой алгоритъм е по-бърз, а сложността по памет – кой алгоритъм заема повече памет в общия случай.

Най-лесния начин да оценим сложността на един алгоритъм е чрез даване на конкретни примери за входни данни, след което можем да изчислим броя операции и нужната памет за тези данни. След това ако удвоим броя данни отново пресмятаме броя операции и нужна памет. Тогава вече имаме информация за изменението на броя операции и заетата памет при изменение на броя входни данни. Дефакто така извеждаме функцията на изменение на времето за изпълнение и обема зета памет спрямо броя на входните данни.

Big O нотация

Най-често използваната нотация за означение на сложност на алгоритми е т.н. Big O нотация. Нейното име идва от начина на изразяване на сложността като главно O, отваряща скоба, функция и затваряща скоба. Например $O(f(n))$, където n е големината на входните данни.

За да намерим точна зависимост на някакъв алгоритъм, можем да изброим броя операции, нужни за изпълнението му при X входни данни. След това при $2X$ входни данни можем да преброим отново операциите и да видим каква е зависимостта.

Накратко ще изброим и опишем основните видове сложност на алгоритми в нарастващ ред, тоест от най-бързите към най-бавните.

Константна сложност – $O(1)$

Константна сложност имат тези алгоритми, които не променят времето за изпълнение и заетата памет при изменение на входните данни. Това са най-бързите алгоритми в общия случай.

Пример за алгоритъм с константна сложност по време, е извеждането на екрана на дължината на масив. Независимо колко на брой клетки има масивът, операцията за изписване на дължината на масива винаги ще отнеме едно и също време.

Пример за алгоритъм с константна сложност по памет е задачата за намиране на максимален елемент в масив – независимо от изменението на броя елементи в масива, на нас ни е нужна само една променлива „мах“, която ни помага да изведем максималния елемент. Тоест паметта, заета от алгоритъма, е константна и не зависи от броя на входните данни.

Логаритмична сложност – $O(\log N)$

Логаритмична сложност имат тези алгоритми, които изпълняват $\log N$ операции при N на брой входни данни. Пример за такъв алгоритъм е следният блок от код:

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
for(int i = 1; i < n; i*=2) {
    System.out.println(i);
}
```

За число N , което представлява обема на данни, които обработваме, масивът изпълнява $\log N$ операции. Това е така, понеже при всяка итерация контролиращата променлива се умножава по две. Алгоритмите с логаритмична сложност също се считат за много, много бързи. Например при $N = 1024$, алгоритъмът ще изпълни десет итерации на цикъла. Ако удвоим N , алгоритъмът ще изпълне 11 операции. Тоест при голямо изменение на входните данни, броя операции се изменя много малко.

Линейна сложност – $O(N)$

Линейна сложност имат тези алгоритми, при които времето и паметта се изменят толкова, колкото се изменя и обема входни данни. Пример за такъв алгоритъм е следният блок от код:

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
```



```
for(int i = 1; i < n; i++) {  
    System.out.println(i);  
}
```

При $N = 1024$, броя итерации също ще е 1024 и програмата ще отпечата 1024 пъти стойността на променливата i . Ако удвоим N , ще се удвоят и броя операции. Ето защо зависимостта е линейна.

Енлог сложност – $O(N \log N)$

Енлог сложността е много често срещана. Най-добрите сортиращи алгоритми са с такава сложност. Прост пример за такава сложност е следния блок от код:

```
Scanner sc = new Scanner(System.in);  
int n = sc.nextInt();  
for(int i = 1; i < n; i++) {  
    for(int j = 1; j < n; j*=2) {  
        System.out.println("i is " + i + " and j is " + j);  
    }  
}
```

Защо сложността тук е $N \log N$? При $N = 1024$ първият цикъл се завърта 1024 пъти, като всеки път вътрешният цикъл се върти 10 пъти. Понеже 10 е логаритъм от 1024, то броя операции за $N = 1024$ е $1024 * \log(1024)$ или $N \log N$ (в нашият случай логаритъмът е двоичен).

Квадратична сложност – $O(N^2)$

Пример за алгоритъм с квадратична сложност е два вложени цикъла, които въртят от едно до N . Тогава броят операции, които ще се изпълнят в тялото на вътрешния цикъл, представлява броя входни данни, повдигнати на квадрат. Примерно за число $N = 10$ броя операции ще е $10 \times 10 = 100$. При удвояване на N , тоест при $N = 20$, броя операции ще се очетвори – $20 \times 20 = 400$.

```
Scanner sc = new Scanner(System.in);  
int n = sc.nextInt();  
for(int i = 1; i < n; i++) {  
    for(int j = 1; j < n; j++) {  
        System.out.println("i is " + i + " and j is " + j);  
    }  
}
```

```
}  
}
```

Кубична сложност – $O(N^3)$

При алгоритми с кубична сложност броят операции представлява броя входни данни, повдигнати на трета степен. Пример за такъв алгоритъм е код с три вложени цикъла. Кубичните алгоритми се считат за бавни.

Експоненциална сложност - $O(2^N)$

Алгоритми с експоненциална сложност се считат за много бавни. Това са такива алгоритми, при които промяната на броя входни данни води до неимоверно нарастване на броя операции. При N входни данни имаме 2^N операции на алгоритъма.

Факториелна сложност – $O(N!)$

Алгоритми с факториелна сложност са едни от най-бавните. В практиката това са сложни задачи, чието решение е много времеемко и досега не е измислена алтернатива с по-малка сложност.



При определяне на сложност на алгоритми, константите във функцията не се взимат предвид. Примерно ако сложността на определен алгоритъм излизе $2N + 31$, то сложността си остава $O(N)$.

Сложност в различни ситуации

Обикновено имаме три вида „сценарии“, при които е добре да оценим сложността на нашите алгоритми:

Сложност при най-благоприятни условия - Best case complexity

Сложността при най-благоприятни условия се изчислява, когато за входни данни вземем най-удобните стойности. Например ако имаме задача да намерим единственото отрицателно число в масив от естествени числа, най-благоприятните входни данни биха били масив, в който отрицателното число да е в началото му. Така при първата итерация числото ще бъде открито и алгоритъмът ще е много бърз, защото сложността ще бъде константна. Независимо колко клетки е масива, броя операции винаги ще е еднакъв.

Сложност при най-неблагоприятни условия - Worst case complexity

Сложността при най-неблагоприятни условия се изчислява, когато за входни данни вземем най-неудобните стойности за конкретните нужди на алгоритъма. В задачата за намиране на единственото отрицателно число в масив от естествени числа, най-неблагоприятните входни данни биха били масив, в който отрицателното число е в края му. Така ще се наложи да се обходи абсолютно целия масив, за да открием отрицателния елемент. Тогава сложността ще бъде линейна, защото броя операции ще е толкова голям, колкото и броя входни данни.

Сложност в общия случай - Average case complexity

Сложността в общия случай се изчислява като вземем за входни данни абсолютно произволна комбинация от стойности и пресметнем средно колко на брой операции ще имаме, когато тези стойности се променят. Обикновено за нас е важно да смятаме сложността на алгоритмите в общия случай.

Глава 11. Сортиране - метод на мехурчето и на пряката селекция.

В тази глава

Метод на мехурчето – основна идея.

Имплементация на алгоритъма.

Оптимизации на алгоритъма.

Анализ на алгоритъма - сложност по време и по памет.

Метод на пряката селекция – основна идея.

Имплементация на алгоритъма.

Оптимизации на алгоритъма.

Анализ на алгоритъма - сложност по време и по памет.

Метод на мехурчето – основна идея.

Методът на мехурчето е лесен за разбиране алгоритъм, който сортира даден масив за квадратично време (N^2). Поради ниската си ефективност, алгоритъмът е подходящ основно за малки масиви с данни. Главната идея на алгоритъма е да сравнява елементите по двойки и да ги разменя ако те са в грешен(несортиран) ред. Това се повтаря за всички елементи и се изпълнява докато нито една размяна не бъде извършена или с други думи, когато масивът вече е не бъде сортиран. Имплементираме алгоритъма чрез два вложени цикъла, като във вътрешния последователно сравняваме двойките съседни елементи и ги разменяме ако не са в правилния ред. На всяка итерация на външния цикъл е гарантирано, че поне един елемент ще си отиде на мястото – а именно – големите елементи постепенно "изплуват" към края на масива. От тук идва и наименованието на алгоритъма. Нека да разгледаме стъпка-по-стъпка как ще се изпълни алгоритъма върху примерен масив от числа.

Примерен масив:

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 1 |
|---|---|---|---|---|

Първа итерация на главния цикъл:

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 1 |
|---|---|---|---|---|

Сравняваме първите два елемента – 1 и 5.

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 3 | 2 | 1 |
|---|---|---|---|---|

Те са в правилния ред, затова проверяваме 5 и 3.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 2 | 1 |
|---|---|---|---|---|

Този път ги разменяме, защото $5 > 3$.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 2 | 1 |
|---|---|---|---|---|

Продължаваме, като сравняваме 5 и 2.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 2 | 5 | 1 |
|---|---|---|---|---|

Отново разменяме, тъй като $5 > 2$.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 2 | 5 | 1 |
|---|---|---|---|---|

Сравняваме 5 и 1.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 2 | 1 | 5 |
|---|---|---|---|---|

Разменяме ги, тъй като $5 > 1$ и 5-та си отива на мястото.

Втора итерация на главния цикъл:

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 2 | 1 | 5 |
|---|---|---|---|---|

Сравняваме първите два елемента – 1 и 3.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 2 | 1 | 5 |
|---|---|---|---|---|

Те са в правилния ред, затова проверяваме 3 и 2.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 5 |
|---|---|---|---|---|

Разменяме ги, тъй като $3 > 2$.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 5 |
|---|---|---|---|---|

Проверяваме 3 и 1.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 1 | 3 | 5 |
|---|---|---|---|---|

Отново разменяме и 3-ата вече си е на мястото.

Трета итерация на главния цикъл:

1 2 1 3 5

Сравняваме 1 и 2.

1 2 1 3 5

Те са в правилния ред, продължава сравнявайки 2 и 1.

1 1 2 3 5

Разменяме ги, тъй като $2 > 1$ и 2-ката вече си е на мястото.

Четвърта итерация на главния цикъл:

1 1 2 3 5

Сравняваме 1 и 1. Няма нужда от размяна.

1 1 2 3 5

Не е имало размяна и алгоритъмът приключва.
Масивът е вече сортиран.

Имплементация на алгоритъма

Имплементацията на алгоритъма е лесна. Това е една и от причините, поради които алгоритъмът е толкова популярен. Алгоритъмът обикаля всички елементи от масива. На всяка итерация проверяваме текущият елемент, заедно със следващият и ако текущият е по-голям от следващия, ги разменяме. Както ще забележим от схемата по-горе, това ни гарантира, че поредния най-голям елемент ще бъде избутан максимално към края на масива. От тук следва, че ако повторим гореописаната операция толкова на брой пъти, колкото са всички елементи, то масивът ще бъде сортиран, тъй като всеки елемент ще си отиде на мястото. Нека да разгледаме имплементацията на алгоритъма:

BubbleSortDemo.java

```
import java.util.Arrays;
public class BubbleSortDemo {
    // метод, който ще сортира подаден като параметър масив
    public static void bubbleSort(int[] num) {
        // повтаряме толкова пъти, колкото са елементите
        for (int i=0; i<num.length; i++) {
            // от първия до предпоследния елемент
            for (int j=0; j<num.length-1; j++) {
                // сравняваме текущия елемент със следващия
                // ако не са в правилната последователност, ги разменяме
                if (num[j] > num[j+1]) {
```

```
        int temp = num[j];
        num[j] = num[j+1];
        num[j+1] = temp;
    }
}

}

public static void main(String[] args) {
    // примерен масив с числа
    int[] num = { 6, 82, 12, 1, 4, 59, -32, 61 };
    bubbleSort(num);
    // извеждаме сортирания масив
    System.out.println(Arrays.toString(num));
}
}
```

Оптимизиране на алгоритъма

Лесно се вижда, че веднъж след като извъртим вътрешния цикъл, най-големия елемент "изплува" в края на масива. Няма смисъл след това да продължаваме да сравняваме другите елементи с него, защото няма да има размяна. Продължавайки тези разсъждения, няма смисъл да сравняваме и с предпоследния елемент след две завъртания на външния цикъл и т.н. От тук следва, че можем да намалим броя елементи, които вътрешния цикъл обикаля, до <броя елементи - 1> - <брой завъртания на външния цикъл>. Това ще оптимизира алгоритъма, макар и общата му сложност да остава същата.

Друга оптимизация, която можем да приложим, е да забележим, че ако масивът в един момент вече е сортиран, няма смисъл вече да правим каквото и да е. Как можем да установим това? Ако масивът е вече сортиран, то ще следва, че във вътрешния цикъл, който разменя два елемента ако те не са в сортирана последователност, няма да се извърши нито една размяна. Можем да си създадем преди този цикъл една булева променлива – флаг, която да ни показва дали имаме поне една размяна, която да се е случила във вътрешния цикъл. Инициализираме тази променлива, със стойност "false". При размяна на елементи задава на тази променлива стойност "true". След края, проверяваме тази

променлива и ако тя е все още "false", то не се е случила размяна и прекратяваме външния цикъл. Можем да видим алгоритъма, реализиран с гореспоменатите оптимизации по-долу:

BubbleSortOptimizedDemo.java

```
import java.util.Arrays;

public class BubbleSortOptimizedDemo {
    // функция, която ще сортира подаден
    // като параметър масив
    public static void bubbleSort(int[] num) {

        // повтаряме толкова пъти, колкото са елементите
        for (int i=0; i<num.length; i++) {
            // дали сме извършили размяна във вътрешния цикъл
            boolean hasASwap = false;
            // от първия до предпоследния елемент минус броя завъртания на
            // горния цикъл
            for (int j=0; j<num.length - 1 - i; j++) {
                // сравняваме текущия елемент със следващия
                // ако не са в правилната последователност, ги разменяме
                if (num[j] > num[j+1]) {
                    int temp = num[j];
                    num[j] = num[j+1];
                    num[j+1] = temp;

                    // отбелязваме, че масива все още не е сортиран
                    hasASwap = true;
                }
            }

            // ако нито една размяна не се е случила
            // то масива вече е сортиран и прекратяваме външния цикъл
            if (!hasASwap) {
                break;
            }
        }
    }
}
```



```
public static void main(String[] args) {  
    // примерен масив с числа  
    int[] num = { 6, 82, 12, 1, 4, 59, -32, 61 };  
    bubbleSort(num);  
  
    // извеждаме сортирания масив  
    System.out.println(Arrays.toString(num));  
}  
}
```

Анализ на алгоритъма - сложност по време и по памет

Очевидно е, че сложността по памет си остава $O(N)$, поради това, че съхраняваме целия масив по време на работа на алгоритъма. Относно сложността по време дори и след предложените оптимизации, алгоритъмът изпълнява два вложени цикъла, всеки от които със сложност $O(N)$, което ни дава обща сложност на алгоритъма по време от $O(N^2)$ в средния и най-лошия случай. Какъв би бил най-добрият случай? С последната оптимизация, за която споменахме по-горе, ако масивът е вече сортиран, то с едно завъртане на вътрешния цикъл ще разберем това и алгоритъмът ще приключи. Това ни дава сложност по време от $O(N)$ в най-добрия случай, а именно – когато масивът е сортиран или е близко до сортиран.

Поради квадратичната сложност по време в най-общия случай, алгоритъмът за сортиране по метода на мехурчето не е подходящ за сортиране на големи обеми от данни.

Метод на пряката селекция – основна идея.

Следващият алгоритъм, който ще разгледаме, също е доста лесен за разбиране, а също и доста популярен. Методът на пряката селекция работи по следния начин – Обхождаме масива, като търсим най-големия елемент. След като го намерим, го поставяме в края на масива. След това обхождаме останалата част от масива в търсене на следващия най-голям елемент. Повтаряме това докато не остане само един елемент. Алгоритъмът отново не е ефективен при особено големи обеми от данни, но е по-бърз от този по метода на мехурчето. Нека да разгледаме стъпка по стъпка как се изпълнява алгоритъма върху примерен масив с пет елемента:

Започваме със следния масив:

| | | | | |
|----|----|----|----|----|
| 33 | 90 | 14 | 26 | 67 |
|----|----|----|----|----|

| | | | | |
|----|----|----|----|----|
| 33 | 90 | 14 | 26 | 67 |
|----|----|----|----|----|

Намираме позицията на най-големия елемент.

| | | | | |
|----|----|----|----|----|
| 33 | 67 | 14 | 26 | 90 |
|----|----|----|----|----|

Разменяме с последния и така той си отива на мястото.

| | | | | |
|----|----|----|----|----|
| 33 | 67 | 14 | 26 | 90 |
|----|----|----|----|----|

Отново намираме най-големия елемент сред останалите.

| | | | | |
|----|----|----|----|----|
| 33 | 26 | 14 | 67 | 90 |
|----|----|----|----|----|

Поставяме го на предпоследна позиция

| | | | | |
|----|----|----|----|----|
| 33 | 26 | 14 | 67 | 90 |
|----|----|----|----|----|

Отново намираме най-големия елемент сред останалите.

| | | | | |
|----|----|----|----|----|
| 14 | 26 | 33 | 67 | 90 |
|----|----|----|----|----|

Разменяме го с текущия последен.

| | | | | |
|----|----|----|----|----|
| 14 | 26 | 33 | 67 | 90 |
|----|----|----|----|----|

Намираме поредния най-голям.

| | | | | |
|----|----|----|----|----|
| 14 | 26 | 33 | 67 | 90 |
|----|----|----|----|----|

Слагаме го на последната останала крайна позиция.

| | | | | |
|----|----|----|----|----|
| 14 | 26 | 33 | 67 | 90 |
|----|----|----|----|----|

Масивът е вече сортиран.

Имплементация на алгоритъма

Как намираме най-големия елемент в масив? Създаваме променлива **maxIndex**, която ще пази индекса на най-големия елемент, който

срещнем при обхождане на масива. Инициализираме променливата със стойност нула – т.е. допускаме, че първият елемент е най-големия. Преглеждаме всички елементи от нулевия до последния, минус тези, които вече са си на мястото(защото иначе последният от тях ще е най-голям) и ако даден елемент е по-голям този на позиция **maxIndex**, то задаваме стойност на **maxIndex** текущия индекс. Така в края на този цикъл в променливата **maxIndex** ще пазим позицията на най-големия елемент. Остава да направим размяна с поредния последен елемент на позиция последната минус броя елементи, които сме наредили в края. Нека да разгледаме примерна имплементация:

SelectionSortDemo.java

```
import java.util.Arrays;

public class SelectionSortDemo {
    static void selectionSort(int[] num) {
        // ще подредим всички елементи без един
        // тъй като той ще си бъде на мястото,
        // след като подредим останалите
        for (int sortedCount=1; sortedCount<num.length;
sortedCount++ ) {
            int maxIndex = 0;
            // намираме индекса на най-големия елемент,
            // като първо допускаме, че е с индекс 0
            // и обновяваме, ако намерим по-голям
            for (int index=0; index<=num.length -
sortedCount; index++) {
                if (num[index] > num[maxIndex]) {
                    maxIndex = index;
                }
            }
            // разменяме като го слагаме
            // на последна позиция минус броя на
            // вече наредените елементи
            int temp = num[maxIndex];
            num[maxIndex] = num[num.length-sortedCount];
            num[num.length-sortedCount] = temp;
        }
    }
}
```

```
public static void main(String[] args) {  
    // примерен масив с числа  
    int[] num = {40, 6, 82, 12, 41, 41, 59, -32, 61};  
  
    selectionSort(num);  
    // извеждаме сортирания масив  
    System.out.println(Arrays.toString(num));  
}
```

Оптимизации на алгоритъма

Най-естествената оптимизация, която можем да приложим наблюдавайки работата на алгоритъма, е докато преглеждаме всички елементи да намираме едновременно най-големия и най-малкия. След това можем да тропаме най-малките в началото, а големите – в края на масива. Ще е необходимо да поддържаме два индекса (текущото начало и край на масива), както и границите, в които на всяка стъпка да търсим поредните най-малък и най-голям елемент. Тази оптимизация би намалила броя завъртания на външния цикъл два пъти, но въпреки това приблизителната сложност на алгоритъма ще си остане същата. Тук трябва да имаме предвид следния проблем – когато разменим най-големият намерен елемент с последния, за да си отиде той на мястото, може да се окаже, че там по случайност стои именно най-малкия. При следваща размяна, когато искаме да преместим най-малкия в началото алгоритъма ще работи некоректно. За да оправим този проблем, ще добавим проверка за това и ще променим индекса на най-малкия в този случай да бъде индекса на най-големия. Нека да видим всичко изложено дотук в следното оптимизирано решение:

SelectionSortOptimizedDemo.java

```

import java.util.Arrays;

public class SelectionSortOptimizedDemo {
    static void selectionSort(int[] num) {
        // за удобство ще си пазим докъде сме подредили
        // в краищата на масива
        int start = 0;
        int end = num.length-1;

        // броим до <броя елементи делено на две>
        // тъй като на всяка стъпка 2 елемента отиват на
        МЯСТОТО СИ
        for (int sortedCount = 0; sortedCount <
        num.length/2; sortedCount++ ) {
            // пазим позициите на най-големия и най-
            малкия

            // като всеки път допускаме че са в началото
            // на несортираната последователност
            int maxIndex = start;
            int minIndex = start;

            // намираме индексите на най-големия и най-
            малкия елемент
            for (int index = start; index <= end; index+
            +) {
                if (num[index] > num[maxIndex]) {
                    maxIndex = index;
                }
                if (num[index] < num[minIndex]) {
                    minIndex = index;
                }
            }
            // разменяме като слагаме най-големия
            // на текущата последна позиция
            int temp = num[maxIndex];
            num[maxIndex] = num[end];
            num[end] = temp;
            // ако в края случайно е бил най-малкия
            // коригираме индекса на най-малкия
            // тъй като вече сме го преместили там

```

където е стоял най-големия

```
        if (minIndex == end) {
            minIndex = maxIndex;
        }
        // слагаме и най-малкия
        // на текущата начална позиция
        temp = num[minIndex];
        num[minIndex] = num[start];
        num[start] = temp;
        // изместваме началото и края на масива
        // като свиваме с по един елемент
        // за да може след това да търсим в
        следващите най-голям и малък в несортираните граници на масива
        end--;
        start++;
    }
}

public static void main(String[] args) {
    // примерен масив с числа
    int[] num = { 6, 82, 12, 41, 41, 59, -32, 61 };
    selectionSort(num);
    // извеждаме сортирания масив
    System.out.println(Arrays.toString(num));
}
}
```

Анализ на алгоритъма - сложност по време и по памет

Лесно се вижда, че алгоритъма извършва квадратичен брой операции по броя елементи на входния масив. Намирането на най-голям/най-малък елемент на всяка стъпка е със сложност по време от порядъка на $O(N)$. Повтаряме това $N/2$ пъти при оптимизирания вариант и $N-1$ пъти при неоптимизирания, където N е броя елементи. От тук и общата сложност на алгоритъма по време е $O(N^2)$. Макар метода на пряката селекция да е със същата сложност като метода на мехурчето, той се представя доста по-добре в средния случай, но въпреки това той си остава неефективен при голям обем от входни данни. Относно сложността по памет – то тя е $O(N)$, защото пазим през цялото време входния масив в паметта.

Глава 12. Сортиране чрез броене.

Radix сортиране.

В тази глава

Сортиране чрез броене – основна идея

Имплементация на алгоритъма

Анализ на алгоритъма

Radix сортиране – основна идея

Имплементация на алгоритъма

Анализ на алгоритъма

Сортиране чрез броене – основна идея

Един от естествените начини да сортираме дадено множество от обекти е да преброим колко пъти се среща всеки от тях. След това можем в нарастващ ред, колкото пъти сме преброили даден обект, толкова пъти да го изведем. Методът на сортирането чрез броене реализира тази идея, като преброява срещанията на елементите и след това ги поставя на правилните им позиции. По време на работата си алгоритъмът използва помощен масив, в който поддържа бройката на всички срещнати елементи.

Въпреки, че алгоритъмът е доста ефективен в общия случай, той не е приложим за числови данни с големи стойности. Защо е така ще разгледаме след малко.

Казахме, че алгоритъмът работи като брои колко пъти се среща всеки елемент. Затова е нужно да знаем колко различни елемента има в масива, като за всеки трябва да пазим колко пъти сме го срещнали. Точно затова имаме нужда от помощния масив. Но каква да е дължината на помощния масив ? Трябва да имаме на разположение толкова клетки, колкото различни числа имаме в оригиналния масив. Ако знаем, че в оригиналния масив имаме числа от 1 до 20, то помощният масив трябва да се създаде с 20 клетки. Понеже ние не знаем предварително какви числа ще има в оригиналния масив, можем просто да намерим максималния елемент в него и да създадем помощен масив с толкова клетки. Тогава ще сме сигурни, че за всяко срещнато число ще имаме налична клетка, в която да пазим колко пъти числото се съдържа в оригиналния масив.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|

Нека имаме следния входен масив: Намираме максималния елемент и той е 3. Създаваме си помощен масив с $3+1 = 4$ клетки, като индексите на клетките ще отразяват стойностите на числата, за които броим колко пъти се съдържат в оригиналния масив. Ще броим срещанията на елементите в следния масив:

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |

Преброяваме първия елемент (3), като в помощния масив увеличаваме броя на тройките.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 0 | 0 | 1 | | | | | | |

Продължава с втория елемент (1), като в помощния масив увеличаваме броя на единиците.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 1 | 0 | 1 | | | | | | |

Срещаме пак единица - отново увеличаваме броя единици в помощния масив.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 1 | 2 | 1 | | | | | | |

Преброяваме и двойката, като на втора позиция увеличаваме броя двойки.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 2 | 1 | 1 | | | | | | |

Продължаваме преброяването с още една единица.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 3 | 1 | 1 | | | | | | |

Аналогично преброяваме и останалите елементи.

...

Накрая имаме за всеки елемент по колко пъти се среща.

| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 4 | 3 | 3 | | | | | | |

Намираме мястото на всеки елемент, като към всяка бройка добавяме броя на елементите преди текущия. Този масив ще сочи най-малката позиция, на която стои вече поставен елемент. Ще четем елементите един по един от масива, ще намаляваме тази позиция, за да се наредят елементите последователно един до друг, и ще поставяме текущия елемент на текущата позиция. Например, ако сега прочетем 2-ка, поглеждаме на 2-ра позиция в този масив, намаляваме позицията там (от 7-ма на 6-та) и съответно на 6-та позиция поставяме елемента със стойност 2. Аналогично по-нататък в масива ако срещнем друг елемент със стойност 2, той ще се нареди на 5-та позиция и така елементите се нареждат отзад напред.

| | | | |
|---|---|---|----|
| 0 | 1 | 2 | 3 |
| 0 | 4 | 7 | 10 |

Обхождаме входния масив и в нов такъв ще поставяме елементите на правилните им места.

Взимаме първия елемент (3) и съответно на трета позиция в масива с позициите, намаляваме текущата позиция. На новополученото място (9) в изходния масив поставяме този елемент.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 4 | 7 | 9 | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |

Продължаваме с втория елемент (1) и съответно на първа позиция в масива с позициите, намаляваме текущата позиция. На това място в изходния масив поставяме този елемент.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 3 | 7 | 9 | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 3 |

По същия начин поставяме и следващия елемент, като отново намаляваме, за да намерим следващата позиция:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 2 | 7 | 9 | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 3 |

По същия начин поставяме и следващия елемент, като отново намаляваме, за да намерим следващата позиция:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 2 | 6 | 9 | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 1 | 0 | 0 | 2 | 0 | 0 | 3 |

Аналогично подреждаме и останалите елементи по техните позиции.

...

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1 | 2 | 1 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | | | | | | |
| 0 | 0 | 4 | 7 | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 |

Имплементация на алгоритъма.

Ще реализираме алгоритъмът точно, както е демонстриран по-горе. За да създадем помощния масив, ще намерим максималният елемент в

оритиналния масив. Броя клетки в помощния масив ще е максималният елемент плюс едно. В този масив ще броим срещанията на всеки елемент. След това ще преправим този масив по описания по-горе начин така, че той да съдържа индексите, на които трябва да стоят елементите. Ще обходим след това за последно входния масив, за да разположим елементите по правилните им позиции в нов изходен масив, използвайки помощния масив, в който сме пресметнали позициите на елементите. Следва имплементацията на алгоритъма:

```
CountingSortDemo.java

package Chapter12;

import java.util.Arrays;

public class CountingSortDemo {
    static int[] countingSort(int nums[]) {

        //намираме най-големият елемент в оригиналния
        масив
        int maxNum = nums[0];
        for(int i = 1; i < nums.length; i++) {
            if(maxNum < nums[i]) {
                maxNum = nums[i];
            }
        }
        // заделяме до maxNum + 1, тъй като масивите се
        индексират от 0
        int numbersCount[] = new int[maxNum+1];
        // тук ще записваме сортирания масив
        int result[] = new int[nums.length];

        // обхождаме всички елементи и преброяваме по
        колко пъти се среща всеки
        for (int index=0; index < nums.length; index++) {
            // вземаме елемента
            int num = nums[index];
```

```
        // на тази позиция увеличаваме броя срещания
        numbersCount[num]++;
    }
    // намираме крайната позиция на всеки елемент
    // като добавяме броя на всички преди него към
неговия брой
    for (int index=1; index < numbersCount.length;
index++) {
        numbersCount[index] += numbersCount[index-
1];
    }
    // тук ще записваме всеки елемент на правилната му
позиция
    for (int index=0; index < nums.length; index++) {
        int num = nums[index];
        numbersCount[num]--;
        int pos = numbersCount[num];
        result[pos] = num;
    }
    return result;
}

public static void main(String[] args) {
    int[] result = countingSort(new int[] {3, 1, 1, 2,
1, 3, 2, 3, 2, 1});
    System.out.println(Arrays.toString(result));
}
}
```

Анализ на алгоритъма

Забелязваме, че сложността на алгоритъма зависи не само от броя елементи, а и от най-големия елемент, тъй като след като ги преброим, трябва да обиколим от нула до най-големия, за да изчислим позициите им в сортирания масив. За да си помогнем в разсъжденията за сложност на алгоритъма, ще въведем две променливи : N – броя елементи и K –

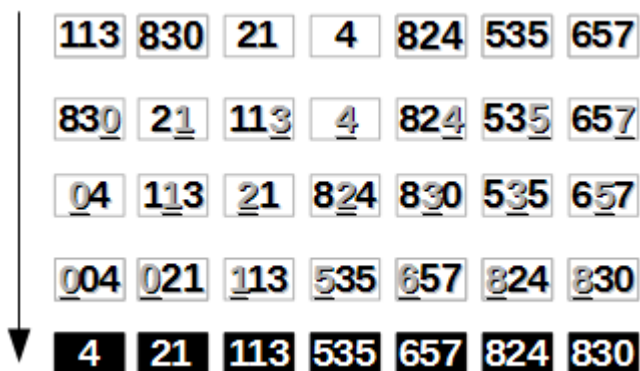
най-големия елемент. От тук алгоритъма не е сложен за анализ по време тъй като се състои от три цикъла. Първият е със сложност $O(N)$, защото обхождаме входния масив, за да преброим срещанията на елементите, втория – със сложност $O(K)$, тъй като обхождаме помощния масив, и накрая третия – със сложност $O(N)$, защото обхождаме отново входния масив. Това ни дава обща сложност на алгоритъма по време от $O(N+K)$. Относно сложността по памет – тъй като имаме входен масив, изходен масив и масив с бройката за всеки елемент, първия и втория заемат $O(N)$ памет, а втория $O(K)$ – то и сложността по памет е от порядъка на $O(N+K)$. От тук следва, че ефективността на алгоритъма много силно се определя от това, кой е най-големия елемент или по-скоро разликата между най-големия и най-малкия. Когато K е много по-малко от N , то алгоритъма става линеен по време и по памет, което го прави наистина много ефективен. Когато, обаче, K е много по-голямо от N , тогава алгоритъмът не е приложим. Например ако трябва да сортираме масив с две клетки, в които едната стойност е 2, а другата – 3 милиона, то тогава ще трябва да си създадем помощен масив с три милиона клетки. След това по време на имплементацията ще обходим този масив, което ще доведе до три милиона итерации. При входни данни от два елемента става ясно, че метода не е ефективен.

Radix сортиране – основна идея

Видяхме, че сортирането с броене е доста добър алгоритъм, когато данните са числа в ограничен диапазон. Но какво ще стане ако имаме N числа в интервала от 0 до N^2 ? Тогава поглеждайки сложността на алгоритъма, ще излезе че сложността по време и по памет ще бъде от порядъка на $O(N^2)$, което ще бъде много неефективно. Как тогава бързо да сортираме числа, които са в по-широк диапазон? А как ще процедираме, ако вместо числа имаме текстови стойности в масива? Можем да използваме алгоритъма за сортиране чрез броене, като част от реализацията на алгоритъма за Radix сортиране. Каква е идеята на Radix алгоритъма? Нека за простота да имаме следните числа, записани в масив:

| | | | | | | |
|-----|-----|----|---|-----|-----|-----|
| 113 | 830 | 21 | 4 | 824 | 535 | 657 |
|-----|-----|----|---|-----|-----|-----|

Ще потредим елементите първо по цифрата на единиците на всеки от тях. След това ще продължим на същия принцип, като вземем за управляваща цифра – цифрата на стотиците. За едноцифрени числа цифрата на стотиците ще сметнем за 0. Продължаваме по същия начин и за цифрата на хилядите и така нататък, докато не преминем през всички значещи цифри на числата.



Стабилни и нестабилни сортиращи алгоритми

Често, когато сортираме някакви елементи, само част от техните свойства се анализират. Например когато сортираме хора по възраст, ние ще имаме предвид само свойството им на колко са години, без да се интересуваме от стойностите на останалите им свойства. Алгоритмите за стабилно сортиране ни гарантират следното: ако два елемента при сравнение са еднакви (като например двама човека на едни и същи години), то тяхната първоначална подредба ще бъде съхранена, така че ако единият от тях е преди другият в началната подредба, то това ще бъде така и в крайния резултат. Нестабилните алгоритми от своя страна не могат да ни гарантират това. Както можем да видим по-горе, за нас е важно алгоритъмът, който използваме на всяка стъпка в радиксната сортировка, да бъде стабилен. Това е така, понеже ако на предишната стъпка сме сортирали по дадена цифра и след това, двата елемента са с равни цифри, то те трябва да си останат в същата последователност. В противен случай ще изгубим сортировката на предишната стъпка.

Имплементация на алгоритъма

Ще модифицираме имплементацията на сортирането чрез броене, за да сортираме само по определена цифра – на единици, десетици, стотици и т.н. Как можем да вземем k-тата цифра на дадено число? Можем да извлечем последната цифра на число като намерим остатъка при деление на 10. Съответно можем да махнем тази цифра като разделим на 10. По същия начин можем да махнем k-1 на брой цифри, разделяйки на 10 на степен k-1. Използвайки това, стигаме до следния метод, който ще ни връща k-тата цифра на дадено цяло число:

```
static int kthDigit(int num, int k) {  
    // намираме 10 на степен k-1, за да махнем първите k-1 цифри  
    int tenOnKth = (int) Math.pow(10, k-1);  
    // делим и връщаме k-тата цифра като резултат  
    return (num / tenOnKth) % 10;  
}
```

От тук нататък можем да използваме този метод, за да сортираме само по дадена цифра. Ще преправим метода за сортиране чрез броене така, че да приеме допълнителен параметър – по коя цифра да се сортират дадените числа. Така за помощен масив няма да имаме нужда от броя на самите елементи, а само на цифрите от 0 до 9 и ще подреждаме елементите, сортирани само по дадена цифра. Имайки това, ще изпълняваме алгоритъма за сортиране чрез броене за всяка цифра от най-малко значещата до последната на най-голямото число в нашата редица. Разбира се преди това трябва да го намерим, както и да преброим с колко цифри е, за да знаем докога да изпълняваме алгоритъма. Споменахме, че помощният алгоритъм, който се използва, трябва да бъде стабилен. Затова ще модифицираме сортирането чрез броене от по-горе така, че да обикаля входния масив в последната си

стъпка отзад-напред. По този начин алгоритъма ще бъде стабилен (защо?). Можем да видим всичко изложено дотук в кода по-долу.

RadixSortDemo.java

```
package Chapter12;

import java.util.Arrays;

public class RadixSortDemo {

    static int kthDigit(int num, int k) {
        // намираме 10 на степен k-1, за да махнем първите
к-1 цифри
        int tenOnKth = (int) Math.pow(10, k-1);
        // делим и връщаме k-тата цифра като резултат
        return (num / tenOnKth) % 10;
    }

    // модифицираме метода за сортиране чрез броене, за да
приема по коя цифра да сортира
    static int[] countingSort(int nums[], int digit) {
        //
        int numbersCount[] = new int[10]; //помощния масив
е от 0 до 9
        int result[] = new int[nums.length];

        for (int index=0; index < nums.length; index++) {
            // вместо числото, ще извлечем определена
цифра от него
            int num = kthDigit(nums[index], digit);
            numbersCount[num]++;
        }

        // намираме крайната позиция на всеки елемент
        // като добавяме броя на всички преди него към
неговия брой
    }
```

```
        for (int index=1; index < numbersCount.length;
index++) {
            numbersCount[index] += numbersCount[index-
1];
        }

        // тук ще записваме всеки елемент на правилната му
позиция
        for (int index=nums.length-1; index >= 0 ;
index--) {
            int num = nums[index];
            // вземаме определена цифра на всяко число
            // и по нея гледаме позицията му в масива
            int numDigit = kthDigit(num,digit);

            numbersCount[numDigit]--;

            int pos = numbersCount[numDigit];

            result[pos] = num;
        }

        return result;
    }

    // метод, който ни връща най-големия елемент в масив
    static int getMaxNum(int nums[]) {
        int max = nums[0];
        for (int index=1; index < nums.length; index++) {
            if (nums[index] > max)
                max = nums[index];
        }
        return max;
    }

    // метод, който ще връща броя цифри на дадено число
```

```

static int getNumOfDigits(int num) {
    int digitCount = 0;
    while (num > 0) {
        num /= 10;
        digitCount++;
    }
    return digitCount;
}

// метод, който сортира масив и го връща като резултат
static int[] radixSort(int nums[]) {
    // намираме максималния брой цифри
    int maxDigits = getNumOfDigits(getMaxNum(nums));
    // сортираме последователно по всички цифри
    for (int digit=1; digit<=maxDigits; digit++) {
        nums = countingSort(nums, digit);
    }
    return nums;
}

public static void main(String[] args) {
    int[] result = radixSort(new int[]{3213, 1322,
123, 221, 9731, 233, 92,23, 9, 85181});

    System.out.println(Arrays.toString(result));
}
}

```

Анализ на алгоритъма

Ще започнем първо с анализа по време. Тъй като radix сортировката използва сортирането чрез броене като част от реализацията си, първо трябва да видим каква ще е сложността на метода, който сортира по дадена цифра. Както знаем вече, сложността на сортирането чрез броене е $O(N+K)$, но в случая $K=10$, защото имаме 10 възможни цифри, по които да подредим, така че сложността е от порядъка на $O(N)$. Тази операция се повтаря \lg_{10} от най-голямото число. От тук следва, че сложността на

алгоритъма по време е от порядъка на $O(N \lg K)$, където K е най-голямото число. Относно сложността по памет, лесно се вижда че създаваме $\lg K$ масива, всеки с големина N , за всяка цифра по която сортираме входната последователност. От там и сложността по памет е $O(N \lg K)$.

Глава 13. Пирамидално сортиране (HeapSort)

В тази глава

Въведение в пирамидата като структура от данни.

Пирамидално сортиране – основна идея.

Имплементация на алгоритъма

Анализ на алгоритъма

Въведение в пирамидата като структура от данни

Преди да говорим за пирамидите, е добре да представим термина „структура от данни“. Структура от данни наричаме такава конструкция, която съхранява съвкупност от стойности по определен ред и според определени правила. Структури от данни са ни нужни, за да организираме голямо количество информация по-лесно. Досега в книгата сме разгледали една структура от данни и това е масивът.

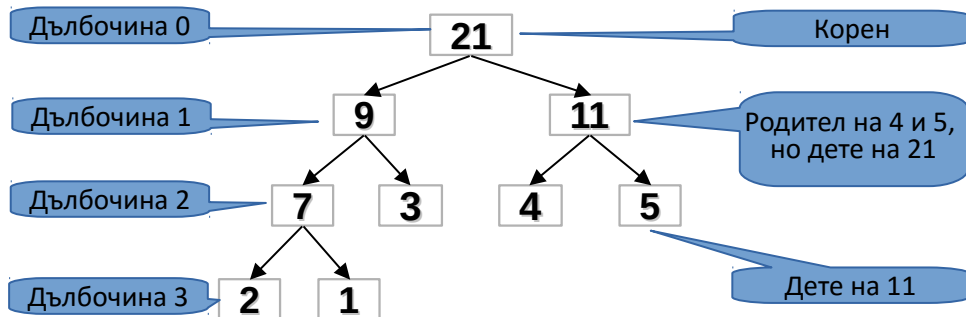
Пирамида е дървовидна структура от данни. Какво значи дървовидна ? Дърветата като структури от данни не са в обхвата на настоящата книга, затова само накратко ще опишем основните характеристики, за да можем по-лесно да си представим пирамидата.

Спомнете си, че масива е линейна структура от данни – елементите са подредени по индекс и според индекса можем да достъпим всеки елемент. При дървовидните структури от данни елементите са разположени в йерархия от типа „родител-дете“, като всеки елемент може да има точно един родител, но много деца. Структурата започва с елемент, който няма родител. Той се нарича корен на дървото. Пътят до даден елемент винаги започва от корена и преминава през неговите деца, докато не се стигне до търсения елемент.

Числа, съхранени в линейна структура от данни – масив:



Числа, съхранени в дървовидна структура от данни:



Дървото има т.н. дълбочина, като горепосоченото дърво е с дълбочина 3. Както казахме, пирамидата е дървовидна структура от данни. Ще разгледаме т.н. бинарна пирамида. Тя отговаря на няколко условия:

- Пирамидата е бинарна структура от данни. Това означава, че всеки родител може да има максимално две деца.
- Пирамидата представлява балансирано дърво. Това означава, че преди да започне да се запълва ново ниво от дървото, предното трябва да е изцяло запълнено с по две деца за всеки родител. Също така запълването на дървото започва от най-лявата част на новото ниво.
- Пирамидата трябва да отговаря на т.н. „пирамидално условие“.

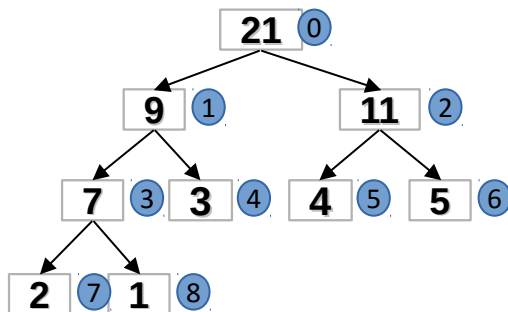
Пирамидално условие

Пирамида наричаме дърво, което отговаря на пирамидалното условие. Пирамидалното условие гласи, че **всеки родител трябва да е със стойност по-голяма от тази на децата си**. Такава пирамида се нарича Мах-пирамида (Max-heap).

Пирамидалното условие може да гласи и обратното – **всеки родител да е по-малък по стойност от децата си**. Тогава пирамидата ще е Min-пирамида (Min-heap).

Реализация на пирамидата с масив

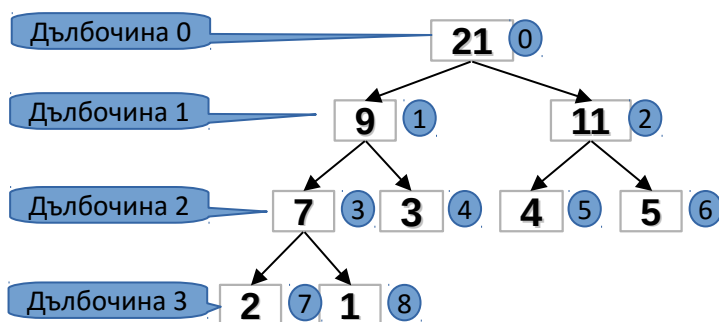
Пирамидата може да се реализира чрез масив. За целта нека визуализираме една пирамида и индексираме всички елементи в нея.



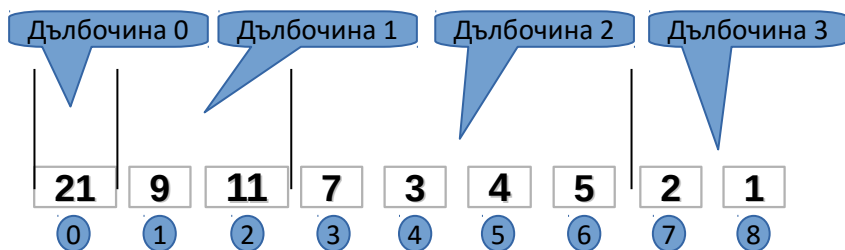
- Ако започнем отгоре надолу и отляво надясно, можем да намерим следната зависимост между индексите на всеки елемент и индексите на неговите деца: за всеки елемент с индекс i , неговото ляво дете е с индекс $2*i + 1$, а неговото дясно - с индекс $2*i + 2$.
- Индексът на всеки родител е $(i-1)/2$, където i е индекса на детето.

Лесно можем да направим проверка на тези формули. Да вземем за пример елемента със стойност 7 от горепосочената пирамида. Индекса на елемента е 3. Неговите деца са с индекси $2*i+1$ и $2*i+2$, за $i=3$ това са елементи с индекс 7 и 8. И наистина, както можем да видим в диаграмата горе, това е така. Родителят на 7-тата е с индекс $(i-1)/2 = 3$. За $i=3$ това е индекс 1 и там седи елемент със стойност 9, който наистина се пада родител на седмицата.

Чрез тези формули можем лесно да представим пирамидата в масив, защото знаем на какви индекси да разположим елементите така, че те да отговарят на пирамидалното условие:



Реализация в масив:

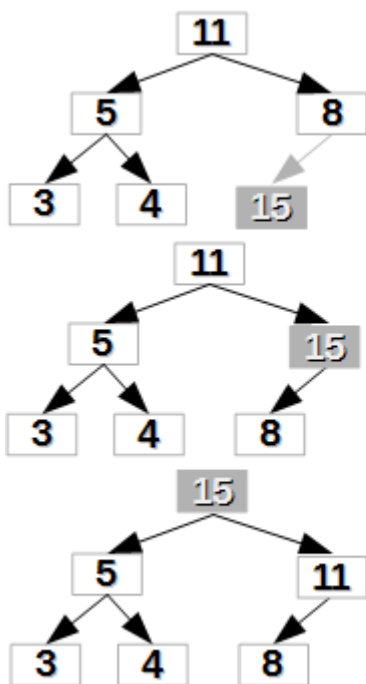


Операции с пирамида

Вмъкване на елемент

Вмъкването на елемент в пирамида изисква т.н. **up-heap** операция или изкачване нагоре по пирамидата. Алгоритъмът за вмъкване на елемент с произволна стойност в пирамидата е следният:

- Добавяме елемента в края на пирамидата – това означава, че елемента става дете в последното ниво на пирамидата.
- Сравняваме стойността на новодобавения елемент с тази на неговия родител. Ако сравнението удовлетворява пирамидалното условие, не правим нищо и алгоритъма приключва.
- Ако сравнението не удовлетворява пирамидалното условие, разменяме елемента с неговия родител и пристъпваме към ново сравняване на елемента и новия му родител (повтаряме т.2).



Добавяме числото 15 към max-heap. Сравняваме с родителя – 8.

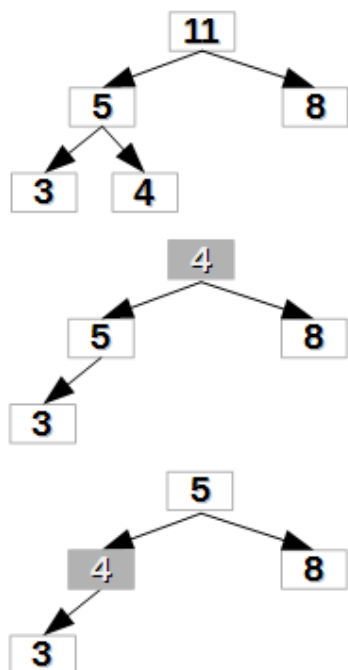
Разменяме ги, защото родителя е по-малък от детето, а това е max-heap. Сравняваме 15 с новият ѝ родител.

Разменяме ги, защото родителя е по-малък от детето, а това е max-heap. 15 вече няма родител и алгоритъмът приключва.

Изтриване на елемент

Изтриване на елемент наричаме операцията по извличането на върха на пирамидата. Понеже вече знаем, че пирамидите отговарят на т.н. пирамидално условие, то върха на една тах-пирамида ще бъде елемента с максимална стойност измежду всички елементи в пирамидата. Аналогично върха на min-пирамида ще бъде елемента с най-малка стойност. Чрез операцията за изтриване, ние реално взимаме максималният или минималният елемент от едно множество. Алгоритъмът е следният:

5. Заменяме върха на пирамидата с последният елемент от последното ниво. След това премахваме последният елемент. По този начин пирамидата остава с елемент по-малко и с нов връх.
6. Сравняваме новият връх с децата му. Ако сравнението удовлетворява пирамидалното условие, прекратяваме алгоритъма.
7. Ако сравнението с някое от децата не удовлетворява пирамидалното условие, разменяме двата елемента и пристъпваме към ново сравняване на елемента и новите му деца (повтаряме т.2).



Премахваме най-големият елемент 11 от max-heap-а. Разменяме с най-отдалеченият елемент – 4.



Сравняваме новият връх с децата му. 4 е по-малко от 5, а това е max-heap, затова ги разменяме.

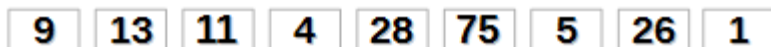


Сравняваме 4 с новите му деца. Удовлетворяват пирамидалното условие и алгоритъмът завършва.

Създаване на пирамида от произволен масив

Създаването на пирамида от произволен масив, наричано още **heapify** процедура, се състои в пренареждане на елементите на масива по такъв начин, че реда им да отговаря на разположението в дървовидна структура, а стойностите им да отговарят на пирамидалното условие.

Да вземем за пример следния произволен масив от стойности:



Как можем да преразпределим елементите на масива така, че да отговарят на условието за max-heap ?

Алгоритъмът е следният:

6. за всеки елемент изпълняваме т.н. down-heap операция. Тя, както описахме по-горе, се състои в следното:
 1. Сравняваме елемента с децата му и ако сравнението не удовлетворява пирамидалното условие, разменяме елемента със съответното дете.
 2. Повтаряме операцията за същия елемент на новата позиция докато елемента не стане връх или докато пирамидалното условие не е изпълнено.

Кодът, представящ алгоритъма е следния:

```
HeapifyDemo.java

import java.util.Arrays;

public class HeapifyDemo {

    public static void main(String[] args) {

        int[] arr = {9,13,11,4,28,75,5,26,1};
        heapifyheapify (arr);
        System.out.println(Arrays.toString(arr));
    }

    public static void heapify( int[ ] arr )
    {
        //за всеки елемент започваме up-heap операция
        for( int i = arr.length - 1; i >= 0; i-- )
            downHeap( arr, i );
    }

    public static void downHeap( int[ ] arr, int index )
    {
        //намираме индексите на децата на елемента
        int leftChild = 2 * index + 1;
```

```

        int rightChild = 2 * index + 2;
        int maxElement = index; //приемаме, че максималният
елемент е на индекс index

        if( leftChild < arr.length && arr[ leftChild ] >
arr[ maxElement ] )
            maxElement = leftChild; //ако лявото дете е с
по-голяма стойност
        if( rightChild < arr.length && arr[ rightChild ] >
arr[ maxElement ] )
            maxElement = rightChild; //ако дясното дете е с
по-голяма стойност
        if( maxElement != index )
        { //разменяме елемента с детето, което е с по-голяма
стойност от него и повтаряме операцията
            int temp = arr[ index ];
            arr[ index ] = arr[ maxElement ];
            arr[ maxElement ] = temp;
            downHeap( arr, maxElement );
        }
    }
}

```

Какво се случва в кода по-горе? В `main` метода инициализираме нашият произволен масив и стартираме метод **heapify**, който да го превърне в пирамида. Методът изпълнява **downHeap** операция за всеки елемент от масива, като започва от елементите в края му. Както знаем вече, елементите в края на масива са от последната дълбочина на пирамидата.

Методът **downHeap** взима индексите на лявото и дясното дете на елемента, за който изпълняваме операция. Сравняваме стойността на децата със стойността на текущия елемент и ако не удовлетворяват условието за `max-heap`, ги разменяме. Ако сме изпълнили размяна, рекурсивно извикваме отново **downHeap** метода, за да повторим операцията докато елемента не си отиде на мястото.

Пирамидално сортиране – основна идея

Алгоритъмът за пирамидално сортиране се основава на идеята, че във всеки момент ние знаем позицията на най-големия елемент в `maxHeap` и можем лесно да го извлечем. За пирамида, представена в масив, максималният елемент (ако е `maxHeap`) или съответно минималният елемент (ако е `minHeap`) е елементът на индекс 0. Извличането на максималният елемент описахме по-горе в раздел „операции с пирамида“ и води до пренареждане на пирамидата така, че тя отново да отговаря на пирамидалното условие. Това означава, че след като извлечем максималния елемент от пирамида и я приведем отново в правилен ред, то отново на нулевия индекс в масива ще имаме следващия по големина елемент.

Стъпките, които трябва да предприемем, за да сортираме произволен масив посредством пирамидална сортировка, са следните:

9. Пренареждаме масива така, че да отговаря на условието за пирамида. Това е т.н. **heapify** операция.
10. Извличаме върха на пирамидата и го разменяме с последния елемент от масива, след което пренареждаме пирамидата посредством **downHeap** операция, като не броим последния елемент от масива. Така ако масивът ни е с N елемента, след първата размяна пирамидата ще се състои от $N-1$ елемента, които ще пренаредим.
11. Повтаряме това N пъти, за да изведем в края на масива всички елементи един по един. По този начин в края на алгоритъма ще сме наредили елементите в нарастващ ред. Ако искаме елементите да са сортирани в намаляващ ред, то тогава **heapify** функцията трябва да преправя масива до **minHeap**.

| | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|-------------------------------------|
| 9 | 13 | 11 | 4 | 28 | 75 | 5 | 26 | 1 | Произволен масив |
| 75 | 28 | 11 | 26 | 13 | 9 | 5 | 4 | 1 | Масивът след heapify операция |
| 1 | 28 | 11 | 26 | 13 | 9 | 5 | 4 | 75 | Разменяме върха с последния елемент |
| 28 | 26 | 11 | 4 | 13 | 9 | 5 | 1 | 75 | Пренареждаме в пирамида остатъка |
| 1 | 26 | 11 | 4 | 13 | 9 | 5 | 28 | 75 | от масива, като игнорираме |
| 26 | 13 | 11 | 4 | 1 | 9 | 5 | 28 | 75 | последният елемент. След това |
| 5 | 13 | 11 | 4 | 1 | 9 | 26 | 28 | 75 | отново повтаряме размяната на |
| 13 | 5 | 11 | 4 | 1 | 9 | 26 | 28 | 75 | първия елемент със следващия по |
| 9 | 5 | 11 | 4 | 1 | 13 | 26 | 28 | 75 | ред „последен“ от пирамидата и |
| 11 | 5 | 9 | 4 | 1 | 13 | 26 | 28 | 75 | така докато не преминем през |
| 1 | 5 | 9 | 4 | 11 | 13 | 26 | 28 | 75 | всички елементи |
| 9 | 5 | 1 | 4 | 11 | 13 | 26 | 28 | 75 | |
| 4 | 5 | 1 | 9 | 11 | 13 | 26 | 28 | 75 | |
| 5 | 4 | 1 | 9 | 11 | 13 | 26 | 28 | 75 | |
| 1 | 4 | 5 | 9 | 11 | 13 | 26 | 28 | 75 | |
| 4 | 1 | 5 | 9 | 11 | 13 | 26 | 28 | 75 | |
| 1 | 4 | 5 | 9 | 11 | 13 | 26 | 28 | 75 | |
| 1 | 4 | 5 | 9 | 11 | 13 | 26 | 28 | 75 | |

Имплементация на алгоритъма

За да реализираме гореописаният алгоритъм, първо трябва да приведем нашият масив в тах пирамида. За целта извикваме метода **heapify**, който подрежда елементите в ред, удовлетворяващ пирамидалното условие. След това в цикъл повтаряме следните операции:

6. разменяме първия с последния елемент от пирамидата
7. намаляваме N, за да може пирамидата да остане с елемент по-малко (пирамидата се състои само от белите квадратчета от горната диаграма)
8. извикваме downHeap метод, който пренарежда пирамидата до правилна структура.

По-долу можем да видим имплементацията на горните разсъждения:

HeapSort.java

```
public class Heapsort {

    private static int N;//броя елементи в масива

    /* Начало на програмата */
    public static void main(String[] args) {
        int[] arr = {9,13,11,4,28,75,5,26,1};
        sort(arr);
        System.out.println("\nElements after sorting ");
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    /* Метод за сортиране */
    public static void sort(int arr[]) {
        heapify(arr);// построяваме пирамида
        for (int i = N; i > 0; i--) {
            swap(arr, 0, i);//разменяме първия елемент
с последния
            N = N - 1;//намаляваме N, за да може
пирамидата да не взима предвид извадения елемент от края на
масива (маркиран с черно в диаграмата)
            downHeap(arr, 0);//пренареждане на остатъка
от масива до изпълнение на пирамидалното условие
        }
    }

    /* Метод за "пирамидизиране" */
    public static void heapify(int arr[]) {
        N = arr.length - 1;
        for (int i = N / 2; i >= 0; i--)
            downHeap(arr, i);
    }
}
```



```

public static void downHeap(int arr[], int i) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int max = i;
    if (left <= N && arr[left] > arr[i])
        max = left;
    if (right <= N && arr[right] > arr[max])
        max = right;

    if (max != i) {
        swap(arr, i, max);
        downHeap(arr, max);
    }
}

/* Метод, който разменя два елемента */
public static void swap(int arr[], int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
}

```

Анализ на алгоритъма - сложност по време и по памет

Сложността на алгоритъма за пирамидално сортиране е $O(N \cdot \log N)$. Това е така, защото имаме цикъл, който се завърта N пъти и за всяка итерация изпълнява `downHeap` операция, която е със сложност $\log N$. `DownHeap` операцията е със сложност $\log N$ поради факта, че дълбочината на пирамидата е $\log N$, където N е броя на елементите ѝ. Следователно когато пренаредим пирамидата така, че на върха ѝ да има елемент, който не отговаря на пирамидалното условие, в най-лошия случай елемента ще бъде сравнен с всички деца в дълбочина. Относно сложността по памет – тъй като имаме входен масив с бройката за всеки

Глава 13. Пирамидално сортиране (HeapSort)

елемент, то той заема $O(N)$ памет. При положение, че не използваме допълнителен масив, то сложността по памет е от порядъка на $O(N)$.

Пирамидалното сортиране се смята за един от най-бързите сортиращи алгоритми.

Глава 14. Сортиране чрез сливане. Бинарно търсене.

В тази глава

Алгоритми от типа „Разделяй и владей“

Сортиране чрез сливане – основна идея.

Имплементация в Java

Анализ на алгоритъма

Двоично търсене – основна идея

Имплементация в Java

Анализ на алгоритъма

Алгоритми от типа „Разделяй и владей“

„Разделяй и владей“ е методология, която се използва в много алгоритми в програмирането. Идеята на такива алгоритми е да се раздели проблема на множество на брой по-малки проблеми, докато не се стигне до такъв проблем, чието решение е тривиално. Решенията на всички разбити по-малки проблеми след това се обединяват, за да се извлече от тях глобалното решение на големия проблем.

Методологията „Разделяй и владей“ е използвана в древността от римляните, за да се справят с опасността от въстания в завладените земи. Те са разселвали завладените народи, като са ги разпределяли на малки части и са ги заселвали в територии, отдалечени от техния дом. По този начин завладяните народи губели силата си, защото числеността и влиянието им в земите намалявали.

В програмирането „Разделяй и владей“ се състои от:

- разбиване на проблема на няколко по-малки части (разделяй)
- решение на по-малките проблеми (владей)
- комбиниране на решенията до постигане на общо решение на проблема

Разделянето на проблема на по-малки части често се осъществява рекурсивно. Разделянето може да се повтаря за всеки по-малък проблем, докато не се стигне до проблем с такъв мащаб, че неговото решение да е тривиално.

Съвкупността от отделни решения на малките проблеми, обаче, не е достатъчна за да кажем, че сме си решили задачата. Нужно е да обединим тези решения, за да получим решение на големия проблем.

Сортиране чрез сливане – основна идея

Сортирането чрез сливане е много добър пример за алгоритъм от типа „Разделяй и владей“. Основната идея на алгоритъма е да разделя масива от числа на по-малки части докато не се получат масиви, които са сортирани. След това всички сортирани масиви се подреждат в по-голям масив, като елементите се подреждат по големина.

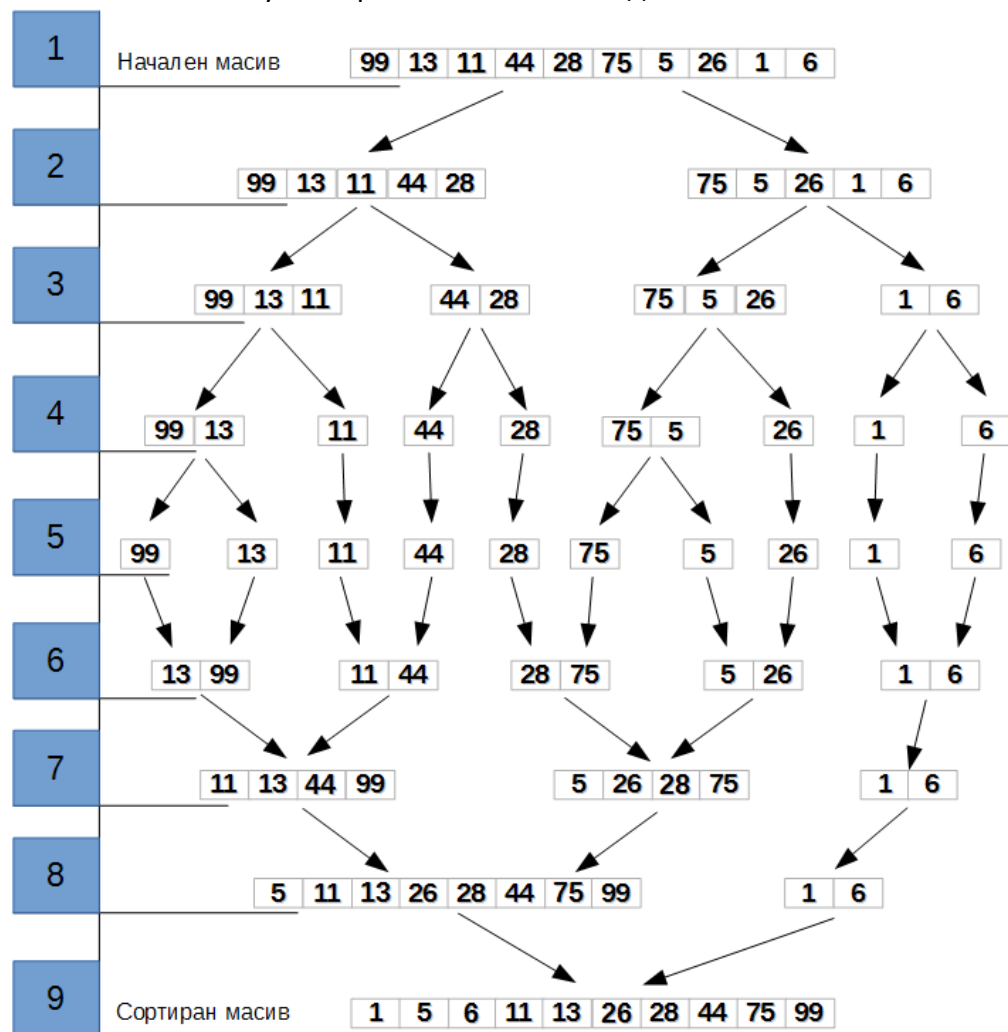
Основния проблем на сортирането чрез сливане е, че имаме масив с N елемента, който трябва да бъде сортиран. Ако разделим масива на две, получаваме два масива с $N/2$ елемента, които трябва да бъдат сортирани. Рекурсивно разделяме и тези масиви на по две и така нататък, докато не получим масиви с по един елемент. Понеже знаем, че всеки масив с един елемент е „сортиран“, то отделните масиви с по един елемент представляват отделни подпроблеми на големия проблем, чиито решения са ясни.

Това, което остава да направим след това, е да комбинираме единичните масиви в по-големи, като нареждаме елементите по големина. Правим това отново рекурсивно, докато не получим масив с оригиналния размер. В него всички елементи вече ще са сортирани.

Нека дадем реален пример с произволен масив от десет елемента. Проблемът ни е, че масивът не е сортиран. Ако разделим масива на два по-малки масива с по пет елемента, ще имаме вече два проблема, но „по-малки“, защото сортирането на 5 елемента изглежда по-лесна задача. А ако продължим и тези два масива от по 5 елемента да ги разделяме на по два нови ? Ще получим масиви от по два и три елемента, които трябва да се сортират. Докога е удачно да делим масивите ? В крайна сметка минималния брой елементи в масив е един. Ако продължаваме да разделяме нашите масиви до толкова, че да се получат много масиви от по един елемент, ще получим много на брой най-малки проблеми. За тези проблеми, обаче, решението е тривиално – масивите с по един елемент са вече сортирани.

Какво направихме всъщност ? Разделихме големия несортиран масив на много малки сортирани масиви. Това е добре, но как да сглобим сортираните масиви така, че да получим големия сортиран масив ? За целта взимаме два вече сортирани масиви и си създаваме нов масив, в

който да обединим елементите. Обединяването на елементите трябва да стане така, че новия масив също да е сортиран. За целта обхождаме паралелно двата взети от нас сортирани масиви и сравняваме елементите един с друг. По-малкият от елементите влиза в новия масив и така докато елементите в двата сортирани масива се изчерпят. Обяснението е визуализирано на схемата по-долу:



- На етап 1 имаме началният несортиран масив. При етап 2 разделяме началния масив на два по-малки несортирани масива. Рекурсивно повтаряме тази операция за новополучените масиви, докато не

достигнем до етап 5, където имаме много масиви, но за тях решението е тривиално – те просто автоматично са сортирани.

- При етап 6 започваме комбиниране на решенията, които сме получили в етап 5. За да комбинираме правилно масивите, трябва новополучените масиви да са сортирани. По този начин отново рекурсивно стигаме до изграждането на новополучен масив с оригиналната големина, но всички елементи в него ще са вече сортирани. Комбинирането на елементи можем да видим подробно на следващата графика. Представено е подробно комбинирането на първите два масива от етап 6



Имплементация в Java

Имплементацията на сиртирането чрез сливане се състои от рекурсивен метод **mergeSort**, който приема като входен параметър несортирания масив, а като резултат връща вече сортиран такъв. В метода има няколко ключови момента:

13. Създаваме два масива с големина – половината от подадения. Всеки от масивите запълваме с елементите на оригиналния масив, като първият запълваме с първата половина от елементи, а във втория слагаме втората половина от елементи. Така получаваме два масива, които представляват разделен на две нашият оригинален масив.
14. Извикваме рекурсивно отново метода, като за входен параметър подаваме всяка от половинките на масива. По този начин рекурсивно всяка половина ще се раздели на още две, които на свой ред ще бъдат разделени на още две и така нататък до получаването на масиви с дължина един елемент.
15. Двата масива се обхождат едновременно с **while** цикъл, като всеки елемент се сравнява с елемент от другия масив и по-малкия влиза първата клетка на оригиналния масив. Това се повтаря докато елементите на един от двата малки масива не се изчерпят.
16. Елементите от неизчерпвания масив се дописват в оригиналния. По този начин реализираме сливането на масиви.

MergeSort.java

```
import java.util.Arrays;

public class MergeSort {

    public static void main(String[] args) {
        int[] arr = { 3, 537, 24, 500,
436,355,3,248,6,449,2,326,322,1};
        System.out.println("unsorted numbers = " +
Arrays.toString(arr));
```



```

        System.out.println("sorted numbers = " +
Arrays.toString(mergeSort(arr)));
    }

    public static int[] mergeSort(int array[])
    {
        if (array.length > 1) {
            int elementsInA1 = array.length / 2;
            int elementsInA2 = array.length -
elementsInA1;

            int arr1[] = new int[elementsInA1];
            int arr2[] = new int[elementsInA2];
            for (int i = 0; i < elementsInA1; i++)
                arr1[i] = array[i];
            for (int i = elementsInA1; i < elementsInA1
+ elementsInA2; i++)
                arr2[i - elementsInA1] = array[i];
            arr1 = mergeSort(arr1);
            arr2 = mergeSort(arr2);

            int i = 0, j = 0, k = 0;
            while (arr1.length != j && arr2.length !=
k) {

                if (arr1[j] < arr2[k]) {
                    array[i] = arr1[j];
                    i++;
                    j++;
                }
                else {
                    array[i] = arr2[k];
                    i++;
                    k++;
                }
            }
            while (arr1.length != j) {
                array[i] = arr1[j];
                i++;
            }
        }
    }

```

```
        j++;
    }
    while (arr2.length != k) {
        array[i] = arr2[k];
        i++;
        k++;
    }
}
return array;
}
```

Анализ на алгоритъма - сложност по време и по памет

Сложността на алгоритъма за сортиране чрез сливане е $O(N \cdot \log N)$. Това е така, понеже при всяка стъпка делим масива на две части, като тези две части на свой ред се делят на по две и т.н. до получаването на масиви с по един елемент. Броя операции на деление на две лесно се пресмята, че е логаритъм втори от N , където N е броя елементи в масива. По време на сливането на масиви за всяка двойка обхождаме елементите, като на едно ниво ние винаги обхождаме всички елементи на оригиналния масив, за да ги слеем в по-големи парчета. Понеже всички елементи са N на брой, имаме N операции по сравнение. Това се случва отново логаритъм от N пъти до получаването на масив с оригиналния размер.

Относно сложността по памет – имаме входен масив с бройката за всеки елемент. При положение, че използваме два допълнителни масива на всяка стъпка, а стъпките са логаритъм от N , то сложността по памет е от порядъка на $O(N \log N)$.

Двоично търсене – основна идея

Двоичното търсене също е алгоритъм от типа „Разделяй и владей“ и е един от най-известните и често използвани методи за търсене на елемент в сортирано множество.

Нека първо изясним проблема, като го обясним по следният начин:

Наш приятел си намисля число от 1 до 100 и ни приканва да го познаем. Когато кажем някое число, той ни отговаря дали сме познали, като ако не сме познали, ни казва дали неговото число е по-голямо или по-малко от предположението ни.

Как ще процедираме ?

Можем просто да започнем от 1, 2, 3 и така нататък, докато не стигнем до числото. Но в този случай ако нашият приятел си е намислил числото 100, ще са ни необходими 100 предположения, докато стигнем до верният отговор, защото след всяко предположение ние елиминираме само една възможна стойност. Тоест в най-лошият случай ако трябва да познаем число в интервала $[1..N]$, ще са ни необходими N предположения.

Двоичното търсене предлага много бърза алтернативна стратегия за търсене на елемент в подредено множество. **Идеята е при всяко предположение да елиминираме половината от възможните стойности.** Нека отново вземем предвид нашата задача да познаем числото от 1 до 100. Можем да предположим, че числото е 50. Тогава след като нашият приятел ни каже дали неговото число е по-голямо или по-малко от нашето предположение, ние автоматично елиминираме половината възможни стойности от интервала. Ако ни се каже, че числото е по-голямо от 50, то оставащите възможни стойности са между 51 и 100. Ако числото е по-малко, то оставащите възможни стойности са между 1 и 49. По този начин само с едно предположение елиминираме половината възможни стойности от интервала. При второто предположение трябва отново да заложим на половината от новия интервал. Така ще постигнем същото – ще елиминираме половината от възможните стойности на новия интервал.

Долната диаграма визуализира разликата между обикновеното линейно търсене и двоичното търсене:

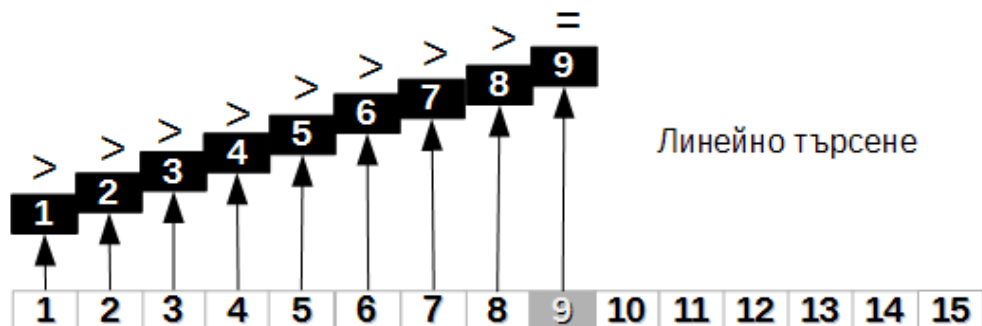
Имаме множество от числа от 1 до 15. Нашият приятел си намисля числото 9.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|

Ако подходим по първия начин чрез започване от началото на списъка, в най-добрия случай, ако нашият приятел е намислил числото едно, с едно

Глава 14. Сортиране чрез сливане. Бинарно търсене.

предположение ще уцелим. В най-лошия случай, обаче, ако нашият приятел е намислил числото 15, ще ни трябват 15 предположения. В нашия случай ще ни трябват 9 предположение.



Ако използваме, обаче, алгоритъмът за двоично търсене, то всеки път, когато дадем предположение, въз основа на отговора ще намаляваме интервала на възможни стойности. По този начин в най-лошия случай ще



имаме логиритъм от N предположения, за да стигнем до намисленото число.

Имплементация в Java

Бинарното търсене се реализира чрез рекурсивен метод. Защо рекурсивен ? Забележете горната диаграма – всеки път след като стесним диапазона от възможни стойности, ние повтаряме една и съща операция. Различни са само началото и края на диапазона. При първото ни предположение възможните стойности са всички – тоест от 0 до N . Създаваме си две променливи – `start` и `end` – те ще съхраняват индексите на първия и последния елемент от диапазона с възможни стойности, между които да познаем. Правим предположение за средата на диапазона – тоест $\text{int mid} = (\text{end} + \text{start}) / 2$. Ако елемента на тази позиция е по-голям от търсеното число, то тогава новият диапазон трябва да е от $\text{mid} + 1$ до `end`. Ако елемента е по-малък, тогава новия диапазон е от `start` до $\text{mid} - 1$.

Рекурсията завършва с две условия за дъно – ако предположението ни е вярно или ако не съществува елемент с такава стойност в масива.

```
BinarySearch.java

import java.util.Arrays;

public class BinarySearch {

    public static void main(String[] args) {
        //инициализираме масива
        int[] array = new int[15];
        //запълваме с числа от 1 до 15
        for(int i = 0; i < array.length; i++) {
            array[i] = i+1;
        }
        int x = 9;
        System.out.println(Arrays.toString(array));
        //извикваме binarySearch метода, като index връща
        индекса на търсения елемент
        int index = binarySearch(array, x, 0,
```

```
array.length-1); //start = 0, end = дължината на масива - 1
    System.out.println(index);
}

    public static int binarySearch(int[] arr, int x, int
start, int end) {
        //нашето предположение е mid елемента
        int mid = (end+start)/2;
        if(start > end) {//ако не съществува елемент в
масива със стойност x
            return -1;
        }
        if(x == arr[mid]){ // ако сме познали числото
            return mid;
        }
        else
            if(x < arr[mid]){ //ако числото е по-малко от
елемента в средата, съкращаваме диапазона
                return binarySearch(arr, x, start, mid-1);
            }
            else
                if(x > arr[mid]){ //ако числото е по-голямо от
елемента в средата, съкращаваме диапазона
                    return binarySearch(arr, x, mid+1, end);
                }
            return -1;
        }
    }
```

Анализ на алгоритъма

Двоичното търсене е много бързо – забележете, че при удвояване на елементите на входния масив имаме само една допълнителна рекурсивна операция. Тоест при масив от 32 елемента ще имаме максимум 6 предположения. При масив от 1024 елемента ще имаме 11 предположения.

Както видяхме, при всяка стъпка алгоритъма съкращава възможните стойности наполовина. Нека направим проста сметка: При масив с четири елемента в най-лошия случай ще имаме три предположения, защото максимум три пъти можем да не отгатнем и съответно да намалим диапазона на половина (4-2-1). При масив с осем елемента – ще имаме максимум четири предположения (8-4-2-1). При масив с шестнадесет елемента – пет предположения (пет пъти съкращаваме диапазона – 16-8-4-2-1) и така нататък. Оттук следва, че сложността на алгоритъма е $\log N + 1$, където N е дължината на масива. Понеже при определяне на сложността на алгоритмите константите се игнорират, излиза, че сложността е $\log N$.

Глава 15. Бързо Сортиране

В тази глава

Основна идея на алгоритъма

Имплементация в Java

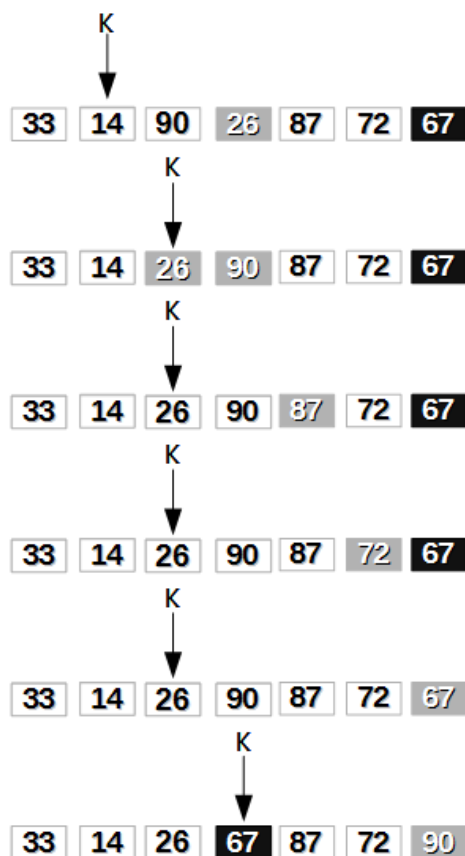
Анализ на алгоритъма

Оптимизации на алгоритъма

Основна идея на алгоритъма

Бързото сортиране е известен алгоритъм, измислен през 1960 г., който сортира елементите за $O(N \lg N)$ време и е един от най-бързите сортиращи алгоритми със сравнение на елементи. Алгоритъмът, освен това, е рекурсивен, т.е. имаме метод, който извиква себе си с по-малък брой елементи и в един момент, когато имаме останал един елемент за сортиране – приключва. На всяка стъпка алгоритъмът фиксира един елемент за разделител (*pivot*) и подрежда останалите така, че отляво да останат по-малките или равни на разделителя, а от дясно – по-големите. За целта използваме една друга променлива *K*, която в началото е равна на -1 и определя на коя позиция ще сложим *pivot*-а след като обходим масива. Така *pivot*-а ще си остане точно на мястото и това, което ни остава, е да сортираме подмасивите от ляво и от дясно на този елемент. Нека да видим как работи разделянето на масива на две части, като фиксираме за **pivot** последния елемент на даден масив:





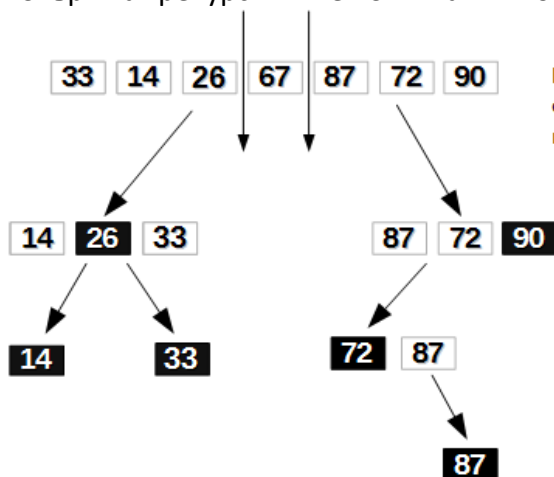
Четвъртият елемент (26) е по-малък от нашия фиксиран, затова увеличаваме числото K и разменяме текущия с K-тия елемент.

Петият елемент (87) е по-голям от нашия фиксиран, затова го оставяме на мястото му.

Шестият елемент (72) е по-голям от нашия фиксиран, затова го оставяме на мястото му.

Накрая разменяме и нашият разделител, като отново увеличаваме K и разменяме с K-тия. По този начин разделителя си отива на мястото и преди него попадат по-малките или равни, а след него – по-големите от него.

След като елемента си дойде на мястото, остава да сортираме по същия начин рекурсивно двете половини на масива – отляво и отдясно на елемента, който вече си е на мястото. Можем да видим накратко как се извършват рекурсивните извиквания по-долу.



Ще приложим рекурсивно бързото сортиране върху двата подмасива отляво и отдясно на елемента-разделител.

Важно е да отбележим, че когато остане само един елемент за сортиране, алгоритъмът приключва работа.

Имплементация в Java

За реализацията ще използваме метод, който ще разделя масива по дадени лява и дясна граница, като за разделител ще вземем най-десният елемент. Методът ще връща позицията, на която сме поставили този елемент. Какво е значението на лявата и дясната граница? Тъй като сортирането ще се изпълнява рекурсивно, ние ще сложим елемент на мястото му и след това ще сортираме само тези вляво от него. След това ще сортираме и тези вдясно. При тези две половини ще подходим аналогично – ще сложим най-крайния елемент на мястото му и след това ще сортираме техните две половини и т.н. Така че за да избегнем копирането на парче от масива, който ще сортираме, ние си поддържаме на всяка рекурсивна стъпка границите от къде до къде сортираме масива. Предаваме на метода за разделяне допълнително тези параметри, за да разделя масива на две части само в дадени граници. Ще използваме тези параметри и като дъно на нашата рекурсия – когато трябва да сортираме даден отрязък от нашия входен масив и лявата и дясната граница съвпадат (т.е. имаме само един елемент) то приключваме с изпълнението на алгоритъма. Как ще реализираме разделянето? Поддържаме си една число "K" – край на областта с по-малки елементи от последния – този който вземаме и за разделител и който е в края на нашия отрязък, който трябва да сортираме. Обхождаме тази област и ако намерим по-малък от нашия разделител, увеличаваме областта на по-малките (числото "K") и разменяме текущия с K-тия, за да може по-малкият да си отиде на мястото. Накрая връщаме числото K, това ще е точно мястото на нашия разделител. Всичко изложено до тук можем да видим в кода по-долу:

`QuickSortDemo.java`

```
import java.util.Arrays;

public class QuickSortDemo {
```

```

// разменя два елемента в масив
static void swap(int a[], int x, int y) {
    int temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}

// взема най-десния елемент в някакъв интервал
// и го поставя на мястото му, като елементите в ляво от
него
// ще са по-малки или равни, а в дясно - по-големи
// като резултат връща мястото на елемента в масива
static int partition(int a[], int left, int right) {
    // запазваме си последния елемент, в края на
интервала
    // той ще е нашия "разделител"
    int pivot = a[right];

    // К показва края на интервала с по-малките
елементи
    // в началото допускаме, че няма по-малки елементи
    // от нашия разделител
    int k = left-1;

    // обхождаме целия интервал
    for (int index=left; index<=right; index++) {
        // ако намерим по-малък от нашия разделител
        if (a[index] <= pivot) {
            // увеличаваме края на интервала

```

```

left..k
// на елементите по-малки от
разделителя
    k++;
// и поставяме текущия при по-малките
// като го разменяме с някой по-голям
    swap(a, k, index);
}
}
// връщаме позицията "K" където стои нашия
разделител
return k;
}

// рекурсивен метод, който сортира масива в граници
// от ляво до дясно
static void quickSort(int a[], int left, int right) {
    // дъното на нашата рекурсия - да имаме 1 или по-
    малко елементи
    // или с други думи - границите да съвпадат
    if (left >= right) {
        return;
    }

    // разделяме масива и запазваме позицията "K" на
    разделителя
    int k = partition(a, left, right);

```

```

        // елемента си е точно на мястото
        // така че сортираме в ляво от него (позиции от
left..k-1)

        // и в дясно (позиции от k+1...right)
        quickSort(a, left, k-1);
        quickSort(a, k+1, right);
    }

    public static void main(String[] args) {
        int a[] = new int[]{3213, 1322, 123, 221, 9731,
233,234,241, 254, 261,239, 92,23, 9, 85181};

        // извикваме метода с масива
        // и в граници от първия до последния елемент
        quickSort(a, 0, a.length-1);
        System.out.println(Arrays.toString(a));
    }
}

```

Анализ на алгоритъма

Бързото сортиране се състои на всяка стъпка от операция за разделяне по даден елемент и две рекурсивни извиквания. Лесно се забелязва, че операцията по разделянето отнема време $O(N)$. Колко пъти изпълняваме това разделяне? Това ще зависи от дълбочината на рекурсията, като в средния случай, когато нашият разделител разделя масива на две сравнително равни части, то ще имаме $\lg_2 N$ рекурсивни извиквания. Това ни дава обща сложност по памет от порядъка на $O(N \lg N)$ в средния случай. Сложността по памет е от порядъка на $O(N)$, тъй като работим върху входния масив през цялото време. Каква би била сложността в най-лошия случай? Да допуснем, че масива е вече сортиран. Тогава ако

вземем последния елемент за разделител, както е в реализацията по-горе, ще имаме по-малките от последния елемент в ляво, но това ще бъдат всички елементи. Освен това няма да имаме дясна част, така че второто рекурсивно извикване няма да подрежда нищо. Така на всяка стъпка по един елемент ще си отива на мястото и ще изпълняваме операцията по разделяне N пъти. Това ни дава сложност на алгоритъма в най-лошия случай от порядъка на $O(N^2)$. Подобен резултат ще получим и ако масива е подреден наобратно.

Оптимизации на алгоритъма

Една от най-лесните оптимизации на алгоритъма, за да избегнем най-лошите случаи е да избираме нашият разделител по произволен начин. Единственото, което ще променим е да вземем произволна стойност използвайки метода **Math.random()**. Този метод ни връща произволно число с плаваща запетая в интервала от 0 до 1, като числото е по-малко от 1.0. Затова ще умножим резултата от този метод по $(right-left+1)$, за да получим интервал от 0 до $(right-left+1)$. Ще съберем със стойността на променливата *left*, за да получим произволно число от *left* до *right+1*, но тъй като метода не връща до 1.0, то ще получим произволно число в интервала от *left* до *right*. Накрая остава да преобразуваме това към цяло число и да разменим с последния елемент. От там нататък метода си остава същия. По-долу можем да видим модифицираната версия на нашия метод "partition".

```
// взема произволен елемент в някакъв интервал
// и го поставя на мястото му, като елементите в ляво от
него
// ще са по-малки или равни, а в дясно - по-големи
// като резултат връща мястото на елемента в масива
static int partition(int a[], int left, int right) {
    // намираме произволен индекс на елемент в
интервала left..right
    int randomIndex = (int)(Math.random() * (right-
left+1)) + left;
```

```

        // поставяме го в края, за да разделяме по него
        swap(a, right, randomIndex);

        int pivot = a[right];

        // К показва края на интервала с по-малките
елементи
        // в началото допускаме, че няма по-малки елементи
        // от нашия разделител
        int k = left-1;

        // обхождаме целия интервал
        for (int index=left; index<=right; index++) {
            // ако намерим по-малък от нашия разделител
            if (a[index] <= pivot) {
                // увеличаваме края на интервала left..k
                // на елементите по-малки от разделителя
                k++;

                // и поставяме текущия при по-малките
                // като го разменяме с някой по-голям
                swap(a, k, index);
            }
        }
        // връщаме позицията "K" където стои нашия разделител
        return k;
    }

```


В тази глава

Задачи за променливи и типове данни

Задачи за бройни системи и побитови оператори

Задачи за условни оператори

Задачи за цикли

Задачи за едномерни масиви

Задачи за двумерни масиви

Задачи за методи

Задачи за рекурсия

Задачи за символни низове

Задачи за сортиращи алгоритми и двоично търсене

Задачи за променливи и типове данни

1. Да се състави програма, която при въведени две числа от конзолата отпечатва на екрана резултата от тяхното събиране, изваждане, умножение и деление, както и остатъка от делението.

Пример:

Въведете стойност на x

8

Въведете стойност на y

5

$x + y = 13$

$x - y = 3$

$x * y = 40$

$x / y = 1$

$x \% y = 3$

2. Да се състави програма, която при подадени три числа от клавиатурата a1, a2 и a3 да размени стойностите им така, че a1 да има стойността на a2, a2 да има стойността на a3, а a3 да има стойността на a1.

3. Да се състави програма, която по въведено трицифрено число от клавиатурата изпише на екрана число с огледална стойност. Това означава, че цифрата на единиците на новото число трябва да е цифрата на стотиците на въведеното, а цифрата на стотиците на новото число трябва да е цифрата на единиците на въведеното.

Пример:

Въведете стойност на x

963

Новото число е 369

4. Да се състави програма, която при въведени две числа x и y разменя стойностите им така, че стойността на x да се предаде на y , а стойността на y да се предаде на x .

Пример:

Въведете стойност на x

5

Въведете стойност на y

9

Преди размяната

x is = 5

y is = 9

След размяната

x is = 9

y is = 5

5. Да се състави програма, която при въведени две булеви числа от конзолата отпечатва на екрана резултата от тяхното логическо И, логическо ИЛИ и логическо ИЗКЛЮЧАЩО ИЛИ.

Пример:

Въведете стойност на x

true

Въведете стойност на y

false

$x \& y$ = false

$x \mid y$ = true

$x \wedge y$ = true

6. Да се състави програма, която при подадено четирицифрено число отпечатва на екрана на отделни редове цифрата на хилядите, стотиците, десетиците и единиците.

Пример:

Въведете стойност на x

4821

Цифра на хилядите = 4
Цифра на стотиците = 8
Цифра на десетиците = 2
Цифра на единиците = 1

7. Да се модифицира първата задача, така че да разменя стойностите на две въведени променливи x и y , но без да се използва трета променлива, а само аритметични операции.

8. Да се състави програма, която по въведени три числа – страни на триъгълник, отпечатва лицето на този триъгълник и неговия периметър.

Пример:

Въведете първата страна:
5
Въведете втората страна:
4
Въведете третата страна:
3
Лицето на триъгълника е:
6
Периметъра на триъгълника е:
12

9. Да се състави програма, която по въведено цяло число отпечатва истина или лъжа(true или false) в зависимост от това дали числото се дели на 2.

Пример:

Въведете стойност:
2134212
Дели ли се на 2:
true

10. Да се състави програма, която прочита 5 булеви стойности (true или false) и извежда истина (true), ако всички от тях имат стойност истина (true).

Пример:

Въведете булевите стойности:

true
true
true
false
true

Всички ли са истина:

false

Задачи за бройни системи и побитови оператори

1. Да се състави програма, която преобразува числото 6 в числото 4 използвайки само побитови операции.

2. Да се състави програма, която въвежда число от 0 до 50 и го извежда в двоичен вид. Да не се използва операцията Integer.toString().

Пример:

Въведете стойност в десетична бройна система:

34

Числото в двоична бройна система:

100010

3. Да се състави програма, която по въведени две цели числа, в трета променлива записва по-голямото от двете. Да се изведе накрая стойността на тази променлива. Да се използват само аритметични и побитови операции.

Пример:

Въведете две стойности:

5

10

По-голямото е:

10

4. Да се състави програма, която въвежда 5 цели числа от конзолата. Дадено е че две двойки от тях са равни, а едно число е различно от останалите. Да се изведе стойността на това число. Да се използват побитови операции.

Пример:

Въведете стойностите:

4

3

6

4

3

Числото без двойник е:

6

5. Да се състави програма, която прочита 3 числа – a , b и c . Да се създаде нова целочислена променлива, на която бъдат вдигнати (да бъдат със стойност 1), битовете на позиции a , b и c .

Пример:

Въведете a , b и c :

0

2

5

Числото е:

41

6. Да се състави програма, която прочита 3 числа – a , b и c . Да се обърнат битовете на позиции b и c в числото a .

Пример:

Въведете a , b и c :

15

0

2

Новото число е:

10

7. Да се състави програма, която по дадени 2 числа да извежда позициите, в които се различават техните битове в двоичното им представяне.

Пример:

Въведете двете числа:

15

10

Битовете се различават в следните позиции:

0 2

8. Да се състави програма, която по дадено цяло число от тип `byte` да извежда `true` ако числото е отрицателно и `false` в противен случай. Да се използват побитови операции за целта.

Пример:

Въведете число:

-10

Отрицателно ли е:

true

9. Да се състави програма, която по дадено цяло число от тип `long` да извежда `true` ако числото е по-голямо от 2^{17} и `false` в противен случай. Да се използват побитови операции за целта.

Пример:

Въведете число:

13183

Отрицателно ли е:

false

10. Да се състави програма, която по дадени 3 цели числа от тип byte да извежда число в двоичен вид, чиито битове са единица на тези позиции, където битовете са единици едновременно в първото и второто число, но не и в третото.

Пример:

Въведете трите числа:

5

7

3

Номера на битове, които са вдигнати в първото и второто, но не и в третото:

100

Задачи за условни оператори

1. Да се състави програма, която по въведено трицифрено число проверява дали числото се дели на всяка своя цифра. Във въведеното число няма да има цифра 0.
2. Да се състави програма, която при въведени три числа от конзолата да ги разпечатва в намаляващ ред.
3. Да се състави програма, която да изведе колко е студено или топло по въведената температура в градус Целзий.

Температурните интервали и съответните съобщения са:

под -20 – ледено студено;

между -20 и -1 – студено

между 0 и 14 – хладно;

между 15 и 25 – топло;

над 25 – горещо

4. Да се състави програма, която приема за вход трицифрено число от вида abc . Програмата трябва да провери следните условия и да изведе съответния изход:

Ако $a = b = c$ – цифрите са равни;

Ако $a > b > c$ – цифрите са във възходящ ред;

Ако $a < b < c$ – цифрите са във низходящ ред;

Ако нито един от гореспоменатите случаи не е налице програмата трябва да изведе "цифрите са ненаредени".

5. Да се състави програма, която по въведени координати на две позиции от стандартна шахматна дъска, извежда отговор дали тези позиции са оцветени в еднакъв или различен цвят.

Пример:

Въведете първа координата на първото поле

2

Въведете втора координата на първото поле

2

Въведете първа координата на второто поле

3

Въведете втора координата на второто поле

2

Двете полета [2,2] и [3,2] са с различен цвят

6. Да се състави програма, която проверява дали въведено четирицифрено число съдържа цифрата 0.

7. Да се състави програма, която по въведени три цели числа - ден, месец и година проверява колко дни са изминали от началото на годината до въведената дата.

8. Трябва да се напълни цистерна с вода. Имате 2 кофи с вместимост 2 и 3 литра и ги ползвате едновременно. Да се състави програма, която по въведен обем извежда как ще прелеете течността с тези кофи, т.е. по колко пъти ще се пълни всяка от кофите.

Пример:

Въведете количество:

52

Ще прелеем по следния начин:

10 пъти с двете кофи и 1 кофа от 2 литра

9. Да се състави програма, която въвежда естествено число от интервала 0..24. Програмата да изведе съответстващо съобщение съобразно въведения час.

Периодите са:

18..4 часа - Добър вечер;

4..9 часа - Добро утро;

9..18 часа - Добър ден

Пример:

Въведете час:

12

Поздрав:

Добър ден

10. Да се състави програма, която по въведени три цели числа - ден, месец и година извежда каква дата ще бъде на другия ден. Да се вземат в предвид броя дни във всеки месец, както и дали годината е високосна или не.

Задачи за цикли

1. Да се състави програма, която при въведени две числа от конзолата изписва всички числа в интервала от по-малкото до по-голямото.

2. Да се състави програма, която при въведено число x от конзолата да извежда първите x числа, които се делят на три без остатък.

3. Да се състави програма, която приема за входни данни естествено число от интервала [1000..99999]. Програмата да отпечата дали това число е палиндром. Палиндром е число, което се изписва по един и

същи начин отпред назад и отзад напред. Примери за палиндром са числата 2772, 12321, 22322 и т.н.

4. Да се състави програма, която прочита число от конзолата и изписва на екрана всички числа, на които това число се дели без остатък.

5. Да се състави програма, която прочита две числа от конзолата и извежда сумата от техните факториели. Факториел на число N е резултата от умножението на всички числа от 1 до N .

6. Да се състави програма, която приема за входни данни датите на раждане на двама човека – Иван и Петър. Програмата да изведе кой от двамата е по-голям, както и разликата във възрастите им. Да се използва цикъл `while`.

Пример:

Дата на раждане на Иван

Въведете ден

12

Въведете месец

3

Въведете година

1987

Дата на раждане на Петър

Въведете ден

17

Въведете месец

7

Въведете година

1987

Иван е роден на 12.3.1987

Петър е роден на 17.7.1987

Петър е по-голям от Иван с 127 дни

7. Да се състави програма, която прочита от конзолата естествено число N . Програмата да извежда първите 10 най-малки числа, които се делят на 2,3 или на 5 и които са по-големи от N . Числата да се подредят заедно с техния пореден номер.

Пример:

Въведете стойност на x

7

1:8 ; 2:9 ; 3:10 ; 4:12 ; 5:14 ; 6:15 ; 7:16 ; 8:18 ; 9:20 ;
10:21

8. Да се състави програма, която извежда квадрат, чиито страни са оформени със знака *, а вътрешността е запълнена с въведен знак. За входни данни се въвежда дължината на страната на квадрата, както и знака, с който да се запълни квадрата.

Пример:

Въведете стойност на дължина на квадрата

9

Въведете символ за запълване

8

```
*****  
*8888888*  
*8888888*  
*8888888*  
*8888888*  
*8888888*  
*8888888*  
*8888888*  
*****
```

9. Да се състави програма, която при въведено число N да изведе като резултат триъгълник с височина N, изграден от знака *.

Пример:

Въведете стойност на x

5

```
  *  
 ***  
*****
```

```
*****
*****
```

10. Да се състави програма, която приема число N в интервала $[1..52]$. Това число отговаря на номер на карта от наредено тесте с карти, като в тестето подредбата е следната:

Наредба по тежест на карта: 2,3,4,5,6,7,8,9,10, Вале, Дама, Поп, Асо.

Наредба по цвят на картите: спатия, каро, купа, пика.

Програмата да извежда картата, която съответства на въведения номер, след което да извежда списък с останалите карти от тестето, подредени от N до 51.

Пример:

Въведете стойност на x

47

Поп купа, Поп пика, Асо спатия, Асо каро, Асо купа, Асо пика

Задачи за едномерни масиви

1. Да се състави програма, която по въведен масив с четен размер да се конструира нов, като половината му елементи са точно като на оригиналния, а другите да са тези елементи, но в обратен ред. Последно, да се изведе новия масив на екрана.

Пример:

Въведете брой елементи:

6

Въведете елементите на масива:

4 2 5 3 1 2

Новият масив:

4 2 5 2 1 3

2. Да се прочете масив и да се отпечата дали е огледален. Даден едномерен масив ще наричаме огледален ако първия елемент е равен на последния, втория на предпоследния и т.н.

Глава 16. Примерни задачи

Следните масиви са огледални:

[3 7 7 3]

[4]

[1 55 1]

[6 27 -1 5 7 7 5 -1 27 6]

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

1 4 6 7 6 4 1

Огледален ли е:

Да

3. Напишете програма, която първо чете масив и после създава нов масив със същия размер по следния начин: стойността на всеки елемент от втория масив да е равна на сбора от предходния и следващият елемент на съответния елемент от първия масив. Стойностите на първия и последния елемент на новия масив са същите като в оригиналния масив. Да се изведе новополучения масив.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

2 5 2 6 1 4 1

Новополученият масив е:

2 4 11 3 10 2 1

4. Напишете програма, която намира и извежда най-дългата редица от еднакви поредни елементи в даден масив.

Пример:

Въведете брой елементи:

10

Въведете елементите на масива:

1 1 1 1 2 2 2 1 1 3

Най-дългата редица от еднакви елементи:

1 1 1 1

5. Да се състави програма, чрез която се въвеждат цели числа в едномерен масив. Програмата да изчислява средната стойност от сумата на елементите, както и да изведе числото, което е най-близко до тази стойност.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

1 2 3 4 5 6 7

Средна стойност:

4

Елемент най-близък до тази стойност:

4

6. Да се състави програма, която въвежда от клавиатурата 7 цели числа в едномерен масив. Програмата да изведе всички числа кратни на 5, но по-големи от 5.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

5 3 1 5 10 -100 200

Търсените елементи са:

10 200

7. Да се състави програма, която въвежда в едномерен масив числа с плаваща запетая. Като изход програма извежда онези 3 различни числа, чиято абсолютна стойност формира максималната обща сума.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

-5.0 0.3 3.2 -8.1 10.1 -8.8 10.1

Търсените елементи са:

10.1 -8.8 -8.1

8. Една редица от естествени числа ще наричаме зигзагообразна нагоре, ако за елементите и са изпълняват условията: $N_1 < N_2 > N_3 < N_4 > N_5 < \dots$. Съставете програма, която проверява и извежда дали въведени в едномерен масив редица от числа изпълняват горните изисквания.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

1 3 2 4 3 7

Зигзагообразна редица ли е:

Да

9. Дадени са два едномерни масива с цели числа. Да се състави програма, която сравнява всички числа с еднакви индекси от двата масива и записва в трети масив, по-голямото от двете числа. Да се изведе новополучения масив на екрана.

Пример:

Въведете брой елементи на първия масив:

3

Въведете брой елементи на втория масив:

3

Елементи на първия масив:

18 3 7

Елементи на втория масив:

7 5 8

Новополученият масив е:

18 5 8

10. Да се състави програма, която по въведено число записва двоичното му представяне в едномерен масив. Да се изведе новополучения масив на екрана.

Пример:

Въведете число:

10

Новополученият масив е:

1 0 1 0

Задачи за двумерни масиви

1. Да се състави програма, която приема за входни данни двумерен масив и го отпечатва в редове и колони, като има празно място между всеки елемент от редовете.
2. Да се състави програма, която за подадена матрица $M \times N$ от числа намира реда, в който сумата от елементите е максимална.
3. Да се състави програма, която за подадена матрица $M \times N$ от числа изчислява сбора от елементите главния и вторичния диагонал и изписва на екрана по-големия от тях.
4. Да се състави програма, която за подадена матрица $M \times N$ от булеви стойности проверява дали се съдържа елемент със стойност true над вторичния диагонал.
5. Да се състави програма, която за подадена матрица $M \times N$ от числа намира произведението на елементите под главния диагонал.

6. Да се състави програма, която по подадена квадратна матрица от естествени числа с размер $M \times M$ проверява дали матрицата е магически квадрат. Магически квадрат наричаме матрица, в която сбора на елементите от всеки ред, всяка колона и всеки диагонал е един и същ.

7. Да се състави програма, която приема за входни данни правоъгълна матрица с числа. Да се намери в нея максималната подматрица с размер 2×2 и да се отпечата на конзолата. Под максимална подматрица се разбира подматрицата, която има максимална сума на елементите, които я съставят.

8. Да се напише програма, която изисква от потребителя да въведе две числа M и N . След това да се построи матрица с размер $M \times N$ по следния начин (примерите са дадени за въведени $N = 4$, $M = 5$).

| | | | | |
|----------|----------|-----------|-----------|-----------|
| 1 | 5 | 9 | 13 | 17 |
| 2 | 6 | 10 | 14 | 18 |
| 3 | 7 | 11 | 15 | 19 |
| 4 | 8 | 12 | 16 | 20 |

9. Да се напише програма, която изисква от потребителя да въведе две числа M и N . След това да се построи матрица с размер $M \times N$ по следния начин (примерите са дадени за въведени $N = 4$, $M = 5$).

| | | | | |
|----------|-----------|-----------|-----------|-----------|
| 1 | 3 | 6 | 10 | 14 |
| 2 | 5 | 9 | 13 | 17 |
| 4 | 8 | 12 | 16 | 19 |
| 7 | 11 | 15 | 18 | 20 |

10. Да се напише програма, която изисква от потребителя да въведе две числа M и N . След това да се построи матрица с размер $M \times N$ по следния начин (примерите са дадени за въведени $N = 4$, $M = 5$).

| | | | | |
|---|---|----|----|----|
| 1 | 8 | 9 | 16 | 17 |
| 2 | 7 | 10 | 15 | 18 |
| 3 | 6 | 11 | 14 | 19 |
| 4 | 5 | 12 | 13 | 20 |

Задачи за методи

1. Да се създаде метод, който приема за параметри масив от цели числа и връща като резултат средноаритметичното от стойностите в масива.
2. Да се създаде метод, който приема за параметри две числа и връща като резултат стринг с информация дали числата са равни по стойност или не.
3. Да се състави метод, който приема масив и принтира масива в конзолата в четим вид.
4. Да се състави метод, който приема за входни данни месец (като стринг) и година (като int) и връща като резултат колко дни съдържа този месец.
5. Да се състави метод, който приема за входни данни два масива от числа с равен брой. Методът да връща като резултат масив, съдържащ произведението на елементите от подадените два масива.
6. Да се състави метод, който приема за входни данни три числа и връща като резултат булева стойност, която отразява дали трите числа могат да бъдат лице на триъгълник.
7. Да се състави метод, който приема за входни данни масив от символи и връща като резултат броя на символите, които са цифри.
8. Да се състави метод, не приема входни данни, а връща като резултат всички числа от 1 до 100, които се делят на 5 без остатък.

9. Да се създаде метод, който приема за параметри два масива с равен брой числа и връща като резултат масив, в който на всяка клетка стои по-големия от елементите в съответните клетки на двата подадени масива.

Пример: При подадени масиви {1,4,8,3} и {3,9,7,5} резултатът трябва да е {3,9,8,5}.

10. Да се създаде метод, който приема за параметри два масива от числа с произволен брой, но подредени в нарастващ ред. Методът да връща като резултат масив с дължина, равна на сбора от дължините на подадените масиви. Стойностите на масива трябва да са същите като стойностите на подадените масиви, но трябва да са подредени в нарастващ ред.

Пример: При подадени масиви {1,4,8} и {3,5,7,9} резултатът трябва да е {1,3,4,5,7,8,9}

Задачи за рекурсия

1. Да се състави програма, която приема за входни данни естествено число N и изписва на екрана стойността на $N!$ (N факториел). Факториел на число N се изчислява чрез умножение на всички числа от 1 до N . Използвайте рекурсия.

2. Да се състави програма, която по въведен масив с естествени числа извежда минималния елемент от масива. Използвайте рекурсия.

3. Да се състави програма, която по въведен масив с естествени числа проверява дали този масив е монотонно нарастващ. Монотонно нарастващ масив е такъв, при който всеки следващ елемент е по-голям от предишния. Използвайте рекурсия.

4. Да се състави програма, която приема за входни данни масив от естествени числа. Програмата да извежда дали в масива има числа, които се повтарят. Използвайте рекурсия.

5. Да се състави програма, която приема за входни данни масив от цели числа. Програмата да извежда на екрана индекса на най-големия елемент. Използвайте рекурсия.

6. Да се състави програма, която приема за входни данни естествено число N . Програмата да извежда число, чиито цифри са написани в ред, обратен на въведеното число. Използвайте рекурсия.

7. Да се състави програма, чрез която по въведени начална и крайна цифра, се извеждат на всеки нов ред редица от числа, започващи с първата цифра и увеличаващи броя си с едно на всеки ред, докато не се стигне до втората – крайна цифра. Използвайте рекурсия.

Пример:

Въведете начална цифра

1

Въведете крайна цифра

6

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

1 2 3 4 5 6

8. Да се състави програма, която проверява дали въведено естествено число е просто. Просто е число, което се дели без остатък единствено на 1 и на себе си. Използвайте рекурсия.

9. Да се състави програма, която приема за входни данни две числа – x и y . Програмата да пресмята произведението на двете числа, като се използва единствено оператора събиране (+). Използвайте рекурсия.

10. Да се състави програма, която приема за входни данни две числа – x и y . Програмата да пресмята стойността на x^y (x на степен y), като се използва единствено оператора събиране (+). Използвайте рекурсия.

Задачи за символни низове

1. Да се състави програма, чрез която се въвеждат два низа, съдържащи до 40 главни и малки букви. Като резултат на екрана да се извеждат низовете само с главни и само с малки букви.

Пример:

Въведете дължина на масивите

4

Въведете буква за масив 1

a

Въведете буква за масив 1

B

Въведете буква за масив 1

c

Въведете буква за масив 1

d

Въведете буква за масив 2

E

Въведете буква за масив 2

f

Въведете буква за масив 2

G

Въведете буква за масив 2

H

ABCD abcd

EFGH efgh

2. Да се състави програма, чрез която се въвежда изречение с отделни думи. Като резултат на екрана да се извежда същия текст, но всяка отделна дума да започва с главна буква, а следващите я букви да са малки.

Пример:

Въведете изречение

супер яКата заДАЧА

Супер Яката Задача

3. Да се състави програма, която чете набор от думи, разделени с интервал. Като резултат да се извеждат броя на въведените думи, както и броя знаци в най-дългата дума.

Пример:

```
Въведете низ от думи
тази задача не е никак лоша
6 думи, най-дългата е с 6 символа
```

4. Да се състави програма, която приема за входни данни низ от символи. Програмата да извежда дали подадения низ е палиндром. Палиндром е такъв низ, който се чете по един и същ начин отляво-надясно и отдясно-наляво.

5. Да се състави програма, която приема за входни данни дума. Програмата да извежда като резултат друга дума, като буквите ѝ са получени като към всеки код на буква от ASCII таблицата е добавено числото 5 и е записана новополучената буква.

Пример:

```
Въведете дума
Hello
Mjqqt
```

6. Да се състави програма, която при въведен текст и въведен символ да отпечатва колко пъти се среща символа в текста.

Пример:

```
Въведете текст
аз обичам баница
Въведете символ
а
Символът се среща 4 пъти
```

7. Да се състави програма, която да извежда ромб от символи. Границите на ромба да са числа, а вътрешността да е запълнена със символ *.

Програмата да приема за входни данни естествено число, което да определя дължината на страна на ромба.

Пример:

Въведете дължина на ромба

4

1

2*2

3***3

4*****4

3***3

2*2

1

8. Да се състави програма, която взима за входни данни два стринга. Всички букви в първия стринг трябва да са различни (неповтарящи се). Програмата да изведе дали двата стринга съдържат еднакви букви и ако не, кои букви от първия стринг не се срещат във втория.

9. Да се състави програма, чрез която се въвежда текст. Програмата да извежда колко пъти е използвана всяка буква от азбуката във въведения текст. Програмата да изведе също и буквата с най-много повторения.

10. Да се състави програма, която по въведен масив от символни низове отпечатва тези низове в рамка. Рамката да е съставена от символа *.

Пример:

Въведете дължина на масива

5

Въведете дума

Hello

Въведете дума

World

Въведете дума

in

Въведете дума

a

Въведете дума


```
frame
*****
*Hello*
*World*
*in    *
*a      *
*frame*
*****
```

Задачи за сортиращи алгоритми и двоично търсене

1. Да се състави програма, която се опитва на всяка стъпка да познае намислено от потребителя число в интервала от 0 до 100. На всяка стъпка програмата извежда предположение за намисленото от потребителя число и като отговор прочита цяло число – 0, 1 или 2. Като съответно 0 означава – надолу, 2 – нагоре, а 1 – числото е познато. Програмата трябва познае числото за не повече от 8 питания, след което да приключи работа.

Пример:

```
30?
2
40?
2
70?
0
60?
1
```

2. Да се имплементира двоично търсене в двумерен масив. Първо въвеждаме размерите и елементите на двумерния масив, след което прочитаем стойността, която търсим. Да се изведе реда и колоната на елемента ако той е намерен или че не е намерен в противен случай.

Пример:

Въведете брой редове:

2

Въведете брой колони:

3

Въведете елементите на масива:

3 2 5

1 8 4

Въведете елемент, който ще се търси:

4

Елемента се намира на ред 2, колона 3.

3. Като се използва операцията за разделяне в бързото сортиране, да се реализира намирането на K-тия най-голям елемент в масив. Т.е. ако е $K=1$, извеждаме най-големия, ако $K=2$ – втория най-голям и т.н. Програмата въвежда масив и числото K , след което извежда търсения елемент.

Пример:

Въведете брой елементи:

8

Въведете елементите на масива:

3 2 5 1 6 9 3 7

Въведете K :

2

Търсеният елемент е: 7

4. Дадени са определен брой думи, да се изведе колко двойки от тях се състоят от еднакви букви.

Пример:

Въведете брой думи:

5

Въведете думите:

баба

абба

аб

ба

петър

Брой двойки думи с еднакви букви: 2

5. Напишете програма, която по дадена последователност от нули и единици ги изкарва сортирани. Да се реализира по възможно най-оптималния начин.

6. Да се сортира даден двумерен масив като се използва алгоритъма за сортиране по метода на мехурчето. Програмата първо прочита масива, а след това го извежда подреден.

Пример:

Въведете брой редове:

2

Въведете брой колони:

3

Въведете елементите на масива:

3 2 5

1 8 4

Масивът сортиран :

1 2 3

4 5 8

7. Да се състави програма, която намира най-често срещания елемент в масив. Да се използва част от сортирането чрез броене.

Пример:

Въведете брой елементи:

15

Въведете елементите на масива:

3 2 3 5 1 5 8 8 4 3 1 4 2 3 3

Най-често срещаният елемент е :

3

8. Да се състави програма, която проверява дали: предварително въведен масив от естествени числа от интервала [0..5000] е сортиран във

възходящ ред. Програмата да извежда и съобщение кои елементи не са в правилна подредба.

Пример:

Въведете брой елементи:

10

Въведете елементите на масива:

1 3 5 7 9 11 10 20 40 300

Масивът не е сортиран.

11 > 10

9. Масив `arr` наричаме хълм (съответно падина), ако за $0 \leq i \leq arr.length-1$, всички елементи с индекси от 0 до i са подредени нарастващо (съответно намаляващо) и всички елементи с индекси от i до `arr.length-1` са подредени намаляващо (съответно нарастващо). Индексът i наричаме екстремум. Напишете метод `int findExtremum(int[] arr)`, който връща екстремума на масива `arr`, който е или хълм или падина. За по-просто допускаме, че масива е с различни елементи. Да се използва двоично търсене.

Пример:

Входен масив:

1 3 5 7 9 11 4 2 1

Екстремум:

11

10. Да се реализира структура от данни пирамида и да се приложи към алгоритъма за сортиране чрез пряка селекция, като на всяка стъпка най-големия елемент да се вади от пирамидата и да се слага в края на нов масив. Да се изведе новополучения сортиран масив на екрана.

Глава 17. Решения

В тази глава

Увод

Как да решаваме задачи по програмиране

Задачи за променливи и типове данни

Задачи за бройни системи и побитови оператори

Задачи за условни оператори

Задачи за цикли

Задачи за едномерни масиви

Задачи за двумерни масиви

Задачи за методи

Задачи за рекурсия

Задачи за символни низове

Задачи за сортиращи алгоритми и двоично търсене

Увод

Ще решим по три задачи от всеки раздел. Понеже задачите в предната глава са подредени по трудност, ще решим първа, пета и десета задача от всеки раздел.

Как да решаваме задачи по програмиране

Винаги трябва да се стремим да измислим алгоритъм на задачата, преди да пристъпим към писането на код.

Важно е да се отбележи, че задачите не трябва да се решават за норматив. Целта на решаването на задачи е да се прихванат и разберат основни похвати за решаване на дребни и тривиални проблеми, след което тези похвати да се преизползват за решението на по-сложните задачи.

Не решавайте задачите за норматив! Не пропускайте задачи! Всяка една задача ще ви обогати, защото ако не ви даде нов похват, то ще ви затвърди вече отработени решения. Алгоритмичното мислене и умението за решаване на проблеми се развиват само чрез много практика. Затова след всяка задача трябва да помислите какво сте научили, какво сте затвърдили и с какво сте подобрили уменията си. Само тогава си е струвало решението.

Решения на задачи за променливи и типове данни

1. Да се състави програма, която при въведени две числа от конзолата отпечатва на екрана резултата от тяхното събиране, изваждане, умножение и деление, както и остатъка от делението.

Пример:

Въведете стойност на x

8

Въведете стойност на y

5

$x + y = 13$

$x - y = 3$

$x * y = 40$

```
x / y = 1
x % y = 3
```

Решение:

За да решим задачата, са ни нужни стойностите на две променливи. Понеже в задачата е казано, че трябва тези стойности да се въведат от конзолата, трябва да използваме конструкцията за четене от конзолата – **Scanner**. Следващата част от задачата е отпечатването на резултатите от основните оператори с променливи. Можем да си заделим по една нова променлива, която да съхранява резултата от всяка операция. След това да отпечатаме стойностите на новите променливи в конзолата. Това ще направим посредством конструкцията за отпечатване в конзолата – **System.out.println(...)**.

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете стойност на x");
    int x = sc.nextInt();
    System.out.println("Въведете стойност на y");
    int y = sc.nextInt();
    int sum = x + y;
    int sub = x - y;
    int mul = x * y;
    int div = x / y;
    int mod = x % y;
    System.out.println("x + y = " + sum);
    System.out.println("x - y = " + sub);
    System.out.println("x * y = " + mul);
    System.out.println("x / y = " + div);
    System.out.println("x % y = " + mod);
}
```

5. Да се състави програма, която при въведени две булеви числа от конзолата отпечатва на екрана резултата от тяхното логическо И, логическо ИЛИ и логическо ИЗКЛЮЧАЩО ИЛИ.

Пример:

Въведете стойност на x

true

Въведете стойност на y

false

x & y = false

x | y = true

x ^ y = true

Решение:

Отново трябва да въведем две променливи, но този път с булева стойност. Затова трябва да ползваме метода nextBool() на скенера, за да можем да прочетем записаното от потребителя и да го присвоим на променлива от тип boolean. След това можем да изпишем резултатите от основните операции с булеви променливи. За разлика от задача 1, обаче, където направихме отделни променливи за резултатите, този път ще запишем резултата от действията директно в конзолата. По този начин си спестяваме излишна памет, която биха заели променливите за резултат.

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете стойност на x");
    boolean x = sc.nextBoolean();
    System.out.println("Въведете стойност на y");
    boolean y = sc.nextBoolean();
    System.out.println("x & y = " + (x & y));
    System.out.println("x | y = " + (x | y));
    System.out.println("x ^ y = " + (x ^ y));
}
```

10. Да се състави програма, която прочита 5 булеви стойности (true или false) и извежда истина (true), ако всички от тях имат стойност истина (true).

Пример:

Въведете булевите стойности:

true

true


```

true
false
true
Всички ли са истина:
false

```

Решение:

Трябва да прочетем от конзолата пет булеви стойности. Това ще направим по вече познатия ни начин – чрез Scanner и конструкцията nextBool(). Как, обаче, да разпознаем дали всички тези пет стойности са true? Нека си припомним основната операция с булеви променливи – И. Операцията И ($x \& y$) връща стойност true ако и двете подадени променливи (x , y) са със стойност true. Ако която и да е от двете променливи е false, то операцията ($x \& y$) ще има стойност false. Този факт ще ни помогне да решим задачата си. Можем да проверим дали първите две от петте въведени променливи са true, ако след като изпълним операция И отговорът е true. В задачата се изисква да изведем на екрана true ако всички променливи са true. Това означава, че дефакто задачата изисква да изведем резултата от логичесното И на всичките пет променливи. Това и ще направим.

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете булевите
стойности");
    boolean x1 = sc.nextBoolean();
    boolean x2 = sc.nextBoolean();
    boolean x3 = sc.nextBoolean();
    boolean x4 = sc.nextBoolean();
    boolean x5 = sc.nextBoolean();
    System.out.println("Всички ли са истина: ");
    boolean result = x1 & x2 & x3 & x4 & x5;
    System.out.println(result);
}

```

Решения на задачи за бройни системи и побитови оператори

1. Да се състави програма, която преобразува числото 6 в числото 4 използвайки само побитови операции.

Решение:

Имаме задача за преобразуване чрез побитови операции. За да можем добре да си представим с какво разполагаме и какво целим, е добре да преобразуваме двете налични числа в двоична бройна система. Числото $6_{(10)}$ в двоична бройна система се изписва като $110_{(2)}$. Числото $4_{(10)}$ се представя като $100_{(2)}$. Сега вече се изправяме пред следния проблем: как от $110_{(2)}$ да изведем $100_{(2)}$? Разликата между двете числа е втория бит – от единица трябва да стане нула. Понеже разполагаме с основните логически оператори и добре знаем до какви резултати води всеки от тях, можем да включим числото $110_{(2)}$ в израз с логически оператор и да помислим с какво число да го комбинираме така, че резултатът да се получи $100_{(2)}$. Ако приложим операция ЛОГИЧЕСКО И, то трябва да направим следното: $110_{(2)} \& XXX_{(2)} = 100_{(2)}$? Познавайки оператора ЛОГИЧЕСКО И можем да заключим, че нужната поредица от единици и нули, която ще доведе до желанния резултат е $100_{(2)}$. Тоест от $110_{(2)}$ за да изведем $100_{(2)}$ трябва да изпълним ЛОГИЧЕСКО И с числото $100_{(2)}$. Но понеже знаем, че числото $100_{(2)}$ е $4_{(10)}$, вече можем да напишем кода.

```
public static void main(String[] args) {
    int x = 6;
    x = x & 4;
    System.out.println(x);
}
```

5. Да се състави програма, която прочита 3 числа – а, b и с. Да се създаде нова целочислена променлива, на която бъдат вдигнати (да бъдат със стойност 1), битовете на позиции а, b и с.

Пример:

Въведете а, b и с:

0

2

5

Числото е:

37

Решение:

Въвеждаме трите променливи по познатия ни начин – чрез скенер и конструкцията `nextInt()`. Това, което трябва да направим, е да конструираме число, което в двоична бройна система има единици за битовете на посочените три позиции. Както знаем, позициите на битовете започват от нула и се броят отлясно наляво. Тоест ако вземем за пример числото $10010_{(2)}$, то има битове със стойност единица на позиции 1 и 4.

И така – как да променим определени битове от нула на единица? Можем да го направим като изпълним операция ЛОГИЧЕСКО ИЛИ с число, което има единица само на този бит, който ни интересува. Тоест ако имам числото $10000_{(2)}$ и искам да направя бит на позиция 2 със стойност 1, то трябва да изпълня логическо ИЛИ с числото $100_{(2)}$, защото това число има бит на позиция 2 със стойност 1 – $10000_{(2)} \mid 100_{(2)} = 100100_{(2)}$. Но кое е това число в десетична бройна система, което има единица само на този бит на позиция две? Това е числото 4. Числото 4 се представя като две на степен 2. Нека разгледаме други числа, степени на двойката и тяхното представяне в двоична бройна система:

| Десетична бройна система | Двоична бройна система |
|--------------------------|------------------------|
| 2 | 10 |
| 4 | 100 |
| 8 | 1000 |
| 16 | 10000 |

Излиза, че всяко число, което е степен на двойката, се представя в двоична бройна система като последователност от нули и точно една единица, която е на позиция, равна на степента на двойката. Тоест число $2^{(N)}$ се представя като число в двоична бройна система с единица на бит на позиция N.

Да се върнем сега на това, което имаме като входни данни и това, което трябва да направим: имаме за входни данни три числа, които представляват позициите на битове, които искаме да са единици. Трябва числото, което има единици на тези битове, да се изведе на екрана. След разясненията дотук стигаме до извода, че е достатъчно да вземем трите числа, които имат единици на позиции a , b и c , тоест трите числа, които са равни на 2^a , 2^b и 2^c и да им приложим операция ЛОГИЧЕСКО ИЛИ. За да намерим стойността на числото две на степен променлива, можем да използваме една от стандартните математически функции, вградени в Java - `Math.pow(p, q)` – резултатът от тази операция е числото p , повдигнато на степен q .

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете a, b и c: ");
    int a = sc.nextInt();
    int b = sc.nextInt();
    int c = sc.nextInt();

    int firstNumber = (int) Math.pow(2, a);
    int secondNumber = (int) Math.pow(2, b);
    int thirdNumber = (int) Math.pow(2, c);
    int result = firstNumber | secondNumber | thirdNumber;
    System.out.println("Числото е ");
    System.out.println(Integer.toBinaryString(result));
    System.out.println(result);
}
```

10. Да се състави програма, която по дадени 3 цели числа от тип `byte` да извежда число в двоичен вид, чиито битове са единица на тези позиции, където битовете са единици едновременно в първото и второто число, но не и в третото.

Пример:

Въведете трите числа:

5
7

3

Число с битове, които са вдигнати в първото и второто, но е и в третото:

100

Решение:

Тук отново трябва да въведем три числа, след което трябва да помислим какви логически операции да използваме върху тях, за да ни доведе резултата от тях до резултата на задачата. Когато решението не е видно при прочитане на задачата, можем да започнем от примера – нека вземем числата 5, 7 и 3 и ги преобразуваме в двоичен вид, след което да поразсъждаваме какво е нужно да направим, за да изведем резултата „100“. След това ще проучим дали това, което сме измислили, ще работи за всички възможни числа.

Приведени в двоичен вид, числата изглеждат така:

| | |
|-----------------|--------------|
| 5 | 1 0 1 |
| 7 | 1 1 1 |
| 3 | 0 1 1 |
| резултат | 1 0 0 |

Виждаме, че трябва да получим число, което има единица на тези позиции, при които в първото и второто число има единица, а в третото – нула. Какви операции са ни нужни, за да стигнем до такова число? Нека разгледаме отделните битове на числата – на позиция две имаме единици в първото и второто число и нула в третото – резултатния бит на тази позиция трябва да е единица. При всички останали случаи резултатния бит трябва да е нула.

Така свеждаме задачата до следната: Да се напише формула, която при подадени три бита **a**, **b**, и **c** да извежда единица само ако **a** и **b** са единица, а **c** е нула. Формулата да извежда нула при всички останали сценарии.

Можем лесно да разберем дали **a** и **b** са единици, като приложим операцията ЛОГИЧЕСКО И на двата бита. Тогава резултата ще е единица

само ако и двете са единици. В противен случай резултата ще е нула. Така вече имаме част от формулата:

```
int x = a & b;
```

Сега трябва да приложим такава операция между x и c , която да извежда единица само ако $x = 1$ и $c = 0$. В противен случай трябва да извежда нула. За целта можем да инвертираме c (да обърнем стойността) и да приложим отново ЛОГИЧЕСКО И. Ако $x = 1$ и $c = 0$, тогава инвертираното c ще бъде равно на 1 и логическото И ще даде единица. При всички останали случаи резултатът ще е нула. Но как се инвертира бит ? Как да обърнем стойността на c ? Това става чрез операция ИЗКЛЮЧАЩО ИЛИ със стойност единица. Защо ? Защото ако $c = 0$, то $0 \wedge 1 = 1$. Ако $c = 1$, то $1 \wedge 1 = 0$. Тоест резултата винаги ще е противоположната стойност на c .

Вече имаме формулата:

```
int x = a & b;
int invertedC = c ^ 1;
int resultBit = x & invertedC;
```

Ако премахнем променливите x и $invertedC$, формулата става така:

```
int resultBit = (a & b) & (c ^ 1);
```

Открихме формула, по която резултатът ще е единица ако първият и вторият бит са единици, а третият е нула. Дотук добре ! Сега трябва да използваме тази формула, за да си решим задачата. Понеже не разполагаме с битове, а с числа, трябва да вземем битовете от всяка позиция на трите числа и поотделно да приложим формулата. Тоест за примерните числа 5,7 и 3 трябва да приложим формулата за три тройки битове – тези на позиции 0, 1 и 2. Но понеже трябва нашата програма да работи за всички числа от тип `byte`, трябва да приложим формулата за всичките осем бита от подадените числа. Оттам ще получим осем резултата, които трябва да обединим в резултатно число. Остават две неща, за които трябва да помислим: как да извлечем всеки отделен бит на съответната позиция от всяко число и как получените резултати да обединим в едно цяло число. Извличането на бит на нулева позиция от число можем да получим, ако приложим ЛОГИЧЕСКО И с единица. Резултатът ще е стойността на първия бит от числото. А как ще вземем втория бит ? Можем да приложим ЛОГИЧЕСКО И с изместена единица наляво един път ($1 \ll 1$, тоест със стойност 10). Резултата можем да го

изместим надясно един път, за да получим втория бит. При третия бит е аналогично, като изместваме съответно два пъти.

А как ще обединим битовете ? Ако имаме резултатни битове 1, 0 и 0, как от тях да направим число ? Решението е – чрез операция ЛОГИЧЕСКО ИЛИ. Но преди това трябва да изместим битовете до съответните им позиции. Така ако изместим единицата две позиции наляво, и първата нула с една позиция наляво, логическото ИЛИ ще ни даде резултат 100.

След всичките тези разсъждения сме готови да облечем нашият алгоритъм в код:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете a, b и c: ");
    byte a = sc.nextByte();
    byte b = sc.nextByte();
    byte c = sc.nextByte();

    //първи бит от числата a, b и c:
    int firstOfA = a & 1;
    int firstOfB = b & 1;
    int firstOfC = c & 1;
    //прилагаме формулата, получена при разсъжденията
и получаваме първия бит на резултата
    int firstOfResult = (firstOfA & firstOfB) &
(firstOfC ^ 1);

    //втори бит от числата a, b и c:
    int secondOfA = (a & 1<<1)>>1;
    int secondOfB = (b & 1<<1)>>1;
    int secondOfC = (c & 1<<1)>>1;
    //получаваме втория бит на резултата
    int secondOfResult = (secondOfA & secondOfB) &
(secondOfC ^ 1);

    //т.н. за останалите битове.
    int thirdOfA = (a & 1<<2)>>2;
    int thirdOfB = (b & 1<<2)>>2;
```

```

        int thirdOfC = (c & 1<<2)>>2;
        int thirdOfResult = (thirdOfA & thirdOfB) &
(thirdOfC ^ 1);

        int fourthOfA = (a & 1<<3)>>3;
        int fourthOfB = (b & 1<<3)>>3;
        int fourthOfC = (c & 1<<3)>>3;
        int fourthOfResult = (fourthOfA & fourthOfB) &
(fourthOfC ^ 1);

        int fifthOfA = (a & 1<<4)>>4;
        int fifthOfB = (b & 1<<4)>>4;
        int fifthOfC = (c & 1<<4)>>4;
        int fifthOfResult = (fifthOfA & fifthOfB) &
(fifthOfC ^ 1);

        int sixthOfA = (a & 1<<5)>>5;
        int sixthOfB = (b & 1<<5)>>5;
        int sixthOfC = (c & 1<<5)>>5;
        int sixthOfResult = (sixthOfA & sixthOfB) &
(sixthOfC ^ 1);

        int seventhOfA = (a & 1<<6)>>6;
        int seventhOfB = (b & 1<<6)>>6;
        int seventhOfC = (c & 1<<6)>>6;
        int seventhOfResult = (seventhOfA & seventhOfB) &
(seventhOfC ^ 1);

        int eighthOfA = (a & 1<<7)>>7;
        int eighthOfB = (b & 1<<7)>>7;
        int eighthOfC = (c & 1<<7)>>7;
        int eighthOfResult = (eighthOfA & eighthOfB) &
(eighthOfC ^ 1);

        int result =      firstOfResult      |
                          secondOfResult <<1 |
                          thirdOfResult  <<2 |
                          fourthOfResult <<3 |

```



```
        fifthOfResult <<4 |  
        sixthOfResult <<5 |  
        seventhOfResult<<6 |  
        eighthOfResult <<7 ;  
        System.out.println("Число с битове, които са  
вдигнати в първото и второто, но е и в третото:");  
  
        System.out.println(Integer.toBinaryString(result));  
    }
```

Решения на задачи за условни оператори

1. Да се състави програма, която по въведено трицифрено число проверява дали числото се дели на всяка своя цифра. Във въведеното число няма да има цифра 0.

Решение:

За да решим задачата, трябва да намерим начин, по който да изведем всяка цифра от трицифреното число и да я запишем в отделна променлива. След това ни остава да проверим дали делението на числото и трите отделни променливи е без остатък.

Нека разпишем трицифреното число като състоящо се от три цифри – abc. Ще направим променлива a, която да пази цифрата на стотиците. Променлива b ще пази цифрата на десетиците, а променлива c – цифрата на единиците. След това ще проверим дали числото се дели без остатък на a, b и c.

Цифрата на стотиците ще намерим, като разделим числото на 100. Понеже делението е целочислено, резултатът ще е точно нужната ни цифра на стотиците.

Цифрата на десетиците ще намерим, като вземем остатъка от делението на сто. Този остатък го делим отново целочислено на 10 и получаваме средната цифра.

Цифрата на единиците получаваме като остатък от делението на цялото число на 10.

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете трицифрено число: ");
    int x = sc.nextInt();// число от тип abc

    if(x < 100 || x > 999) {
        System.out.println("Не сте въвели трицифрено
число.");
    }
    else {
        int a = x/100;//стотици
        int b = x%100/10;//десетици
        int c = x%10;//единици
        if(a == 0 || b == 0 || c == 0) {//проверяваме
дали някоя цифра не е нула
            System.out.println("Въведеното число
съдържа нула.");
        }
        else {
            if(x % a == 0 && x % b == 0 && x % c == 0)
{//проверяваме дали числото се дели на всяка от цифрите му
без остатък
                System.out.println("Числото се дели на
всичките си цифри без остатък");
            }
            else {
                System.out.println("Числото не се дели
на всичките си цифри");
            }
        }
    }
}
```

5. . Да се състави програма, която по въведени координати на две позиции от стандартна шахматна дъска, извежда отговор дали тези позиции са оцветени в еднакъв или различен цвят.

Пример:

Въведете първа координата на първото поле

2

Въведете втора координата на първото поле

2

Въведете първа координата на второто поле

3

Въведете втора координата на второто поле

2

Двете полета [2,2] и [3,2] са с различен цвят

Решение:

Нека разгледаме шахматната дъска, заедно с координатите на полетата ѝ:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | black | white | black | white | black | white | black | white |
| 1 | white | black | white | black | white | black | white | black |
| 2 | black | white | black | white | black | white | black | white |
| 3 | white | black | white | black | white | black | white | black |
| 4 | black | white | black | white | black | white | black | white |
| 5 | white | black | white | black | white | black | white | black |
| 6 | black | white | black | white | black | white | black | white |
| 7 | white | black | white | black | white | black | white | black |

Нека вземем координатите на няколко бели полета: [1,0], [1,3], [3,2], [5,4],[6,3] и т.н.

Нека сега вземем координатите на няколко черни полета: [1,1], [1,5], [4,2], [5,7], [7,3] и т.н.

Какво забелязваме ? На пръв поглед няма конкретна зависимост между координатите на полетата. Това, обаче, което прави впечатление, когато си поиграем малко с числата, е, че сбора на координатите на белите полета винаги е нечетно число, докато сбора на координатите на черните полета винаги е четно. Можем да използваме този факт, за да разберем дали координатите на две полета са с различен цвят. Ако сбора на координатите на едното поле е четен, а сбора на координатите на другото поле е нечетен, значи полетата са с различен цвят. За да разберем дали стойността на една променлива е четна или нечетна, трябва да видим дали остатък от делението на две е равен на нула или на едно. При остатък 0 променливата съдържа четна стойност. При остатък 1 – променливата съдържа нечетна стойност.

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете първа координата на
първото поле: ");
    int x1 = sc.nextInt();
    System.out.println("Въведете втора координата на
първото поле: ");
    int y1 = sc.nextInt();
    System.out.println("Въведете първа координата на
второто поле: ");
    int x2 = sc.nextInt();
    System.out.println("Въведете втора координата на
второто поле: ");
    int y2 = sc.nextInt();

    int sum1 = x1+y1;//сбора от координатите на поле 1
    int sum2 = x2+y2;//сбора от координатите на поле 1
    //за по-добра четимост ще изведем булеви променливи за
различните сценарии
    boolean different1 = (sum1 % 2 == 0) && (sum2 % 2 !=
0);//сума 1 е четна,а сума2 - нечетна
    boolean different2 = (sum1 % 2 != 0) && (sum2 % 2 ==
```

```

0); //сума 1 е нечетна, а сума2 - четна

    if( different1 || different2){
        System.out.println("Двете полета ["+x1+", "+y1+"]
и ["+x2+", "+y2+"] са с различен цвят");
    }
    else {
        System.out.println("Двете полета ["+x1+", "+y1+"]
и ["+x2+", "+y2+"] са с еднакъв цвят");
    }
}

```

10. Да се състави програма, която по въведени три цели числа - ден, месец и година извежда каква дата ще бъде на другия ден. Да се вземат в предвид броя дни във всеки месец, както и дали годината е високосна или не.

Решение:

В тази задача проблемните сценарии са три – 28 февруари, 13 декември и 30-то число на месеца. При всички други ситуации единствено ще инкрементираме цифрата на деня. Нека разгледаме трите проблемни сценария:

Ако деня ни е числото 30, в някои месеци ще сме стигнали края на месеца, а в други ще имаме още един ден. Тогава трябва да проверим дали числото на месеца ни отговаря на месец с 30 дни или с 31 дни. Ако месеца е с 30 дни, инкрементираме числото на месеца, а числото на деня става 1.

Ако числото е 28 и месеца е февруари, трябва да проверим дали годината е високосна. Проверката за високосна година е следната:

- годините, кратни на 4 са високосни, останалите не са;
- изключение 1: годините, кратни на 100 не са високосни;
- изключение 2: годините, кратни на 400 са високосни.

При високосна година инкрементираме деня, а при невисокосна – връщаме първи март.

Ако числото е 31 и месеца е декември, трябва да преминем към следващата година.

Това са сценариите, които трябва да обслужим. Добре е да имаме валидация за данните в датата – тоест да не допускаме потребителя да въведе число за месеца, по-голямо от 12 и число за деня, по-голямо от 31.

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете ден: ");
    int day = sc.nextInt();
    System.out.println("Въведете месец: ");
    int month = sc.nextInt();
    System.out.println("Въведете година: ");
    int year = sc.nextInt();

    boolean leapYear = (year % 4 == 0 && (year % 400
== 0 || year % 100 != 0)); // дали годината е високосна

    // валидация на данните:
    boolean correctData = true;
    if (day < 0 || day > 31) {
        correctData = false;
    } else if (month < 0 || month > 12) {
        correctData = false;
    } else if (year < 0) {
        correctData = false;
    } else if (day == 29 && month == 2 && !leapYear) {
        correctData = false;
    }

    if (day == 28 && month == 2) { // при 28 февруари
        // проверяваме дали годината е високосна
        if (leapYear) {
            day++; // при високосна година имаме
29ти февруари
        } else { // в противен случай - 1ви март.
```

```

        day = 1;
        month++;
    }
} else
    // ако въведената дата е 29.2 и годината е
високосна, правим датата на 1ви март
    if (day == 29 && month == 2 && leapYar) {
        day = 1;
        month++;
    } else
        // ако е въведен ден 30ти, увеличаваме деня за
всички месеци, които са с 31 дни.
        if (day == 30) {
            switch (month) {
                case 1:
                case 2:
                    correctData = false;
                    break; // 30ти февруари е невалидна
дата

                case 3:
                case 5:
                case 7:
                case 8:
                case 10:
                case 12:
                    day++;
                    break;

                // за всички месеци с 30 дни деня става
1ви, а месеца - следващия.
                case 4:
                case 6:
                case 9:
                case 11:
                    day = 1;
                    month++;
                    break;
            }
}

```

```

        } else
            // ако е въведен ден 31ви, преминаваме на
            следващия месец, като единствено при декември преминаваме на
            следващата година.
            if (day == 31) {
                if (month == 2) {
                    correctData = false; // 31ви февруари е
невалидна дата
                }
                day = 1; // рестартираме деня
                if (month == 12) { // ако месеца е декември,
рестартираме месеца и вдигаме годината с 1
                    month = 1;
                    year++;
                } else { // в противен случай вдигаме месеца
с 1
                    month++;
                }
            } else { // за всяка друга стойност на деня
единствено вдигаме деня с едно и не бутаме месеца и годината
                day++;
            }

            if (correctData) {
                System.out.println("Следващата дата е");
                System.out.println(day + "." + month + "."
+ year);
            } else { // ако в кода някъде имаме невалидни
данни за въведена дата, то тогава променливата correctData е
false.

                System.out.println("Невалидна дата");
            }
        }
    }

```

Това решение е доста дълго, но конструкциите в него са добре известни и разбираеми. Съществуват и други решения на задачата, при които кода е по-кратък, но дали е по-четим и разбираем, зависи от гледната точка. Нека представим втори вариант на решението на задачата. За целта се

оповаваме на великата математика и зависимостите между цифрите в датите.

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете ден: ");
    int day = sc.nextInt();
    System.out.println("Въведете месец: ");
    int month = sc.nextInt();
    System.out.println("Въведете година: ");
    int year = sc.nextInt();

    // дали годината е високосна
    boolean leapYear = (year % 4 == 0 && (year % 400
== 0 || year % 100 != 0));

    // валидация на данните
    boolean correctData = true;
    if (day < 0 || day > 31) {
        correctData = false;
    } else if (month < 0 || month > 12) {
        correctData = false;
    } else if (year < 0) {
        correctData = false;
    } else if (day == 29 && month == 2 && !leapYear){
        correctData = false;
    }

    int daysInMonth = 31;
    int numOfMonth = 12;

    // ако погледнем броя дни в месеците, виждаме че
    те са 31 като в четните месеци до 7-мия включително имаме 30
    дни, а в нечетните от 7-мия нататък пак имаме 30 дни
    if ( (month % 2 == 0 && month <= 7) || (month % 2
== 1 && month > 7) ) {
        daysInMonth = 30;
    }
}
```

```

        // а във февруари имаме 28 или 29 дни
        if (month == 2) {
            daysInMonth = 28;
            if (leapYear) {
                daysInMonth++;
            }
        }

        day++;

        // увеличаваме месеца като ако дните са минали
        броя дни в месеца ще получим 1 примерно : 32 ден / 31 дена в
        месеца = 1 и това ще добавим към месеца
        month += day / daysInMonth;

        // ако сме преминали броя дни в месеца с остатък
        при деление на броя дни ще получим 1-ви ден от новия месец в
        противен случай деня ще си остане същия примерно 32 ден % 31
        = 1, т.е. първи ден от новия месец
        day %= daysInMonth;

        // по същия начин процедираме и с годината и
        месеца

        year += month / numOfMonths;
        month %= numOfMonths;

        if (correctData) {
            System.out.println("Следващата дата е");
            System.out.println(day + "." + month + "."
+ year);
        } else { // ако в кода някъде имаме невалидни
данни за въведена дата, то тогава променливата correctData е
false.

            System.out.println("Невалидна дата");
        }
    }
}

```

Решения на задачи за цикли

1. Да се състави програма, която при въведени две числа от конзолата изписва всички числа в интервала от по-малкото до по-голямото.

Решение:

Нека запишем въведените от потребителя числа в две променливи *a* и *b*. Трябва да изпишем всички числа в интервала от по-малкото, към по-голямото. Това означава, че трябва да изпълним конструкцията за изписване на число в конзолата *n* пъти, като *n* е разликата между *a* и *b*. Понеже трябва да повторим една и съща конструкция много на брой пъти, първото нещо, за което трябва да се сетим, е да изпълним цикъл, в чието тяло да използваме нужната конструкция. Колко пъти ще се върти цикъла ? От по-малкото число до по-голямото. Дотук добре. Но кое от двете числа е по-малкото ? Не знаем. Значи трябва да проверим. Защо е нужна проверката ? Ами ако завъртим един цикъл от *a* до *b* и се окаже, че потребителят въведе за *a* по-голяма стойност, отколкото за *b*, тогава цикълът няма да има нито една итерация. Затова първото нещо, което трябва да направим, е да проверим коя стойност е по-голяма. Нека се опитаме да резлизираме описаното:

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число a");
    int a = sc.nextInt();
    System.out.println("Въведете число b");
    int b = sc.nextInt();

    if(a > b ) {
        for(int i = b; i <= a; i++){
            System.out.print(i + " ");
        }
    } else {
        for(int i = a; i <= b; i++){
            System.out.print(i + " ");
        }
    }
}
```

```

        }
    }
}

```

Вече сме решили задачата. Но нещо не е наред. Повтаря се код. Забележете, че и в тялото на `if`-а и в `else`-а се прави едно и също – върти се цикъл, който принтира числа от едно число до друго. Винаги е добра практика да не се повтаря код във вашите програми. Това е така, защото при повтаряне на код програмите стават по-трудно поддържани. Например ако трябва да променим решението на задачата да отпечата всички числа, но през едно, то тогава ще трябва да направя редакция на кода и на двете места. Ако повтаряме код 5 или 10 пъти, редакцията трябва да се направи на всичките 5 или 10 места. Ако забравим някъде да поправим кода, програмата ще се счупи. Затова е добре да помислим как да резлизираме решение без повтаряне на код. В текущата задача може да ви се стори безсмислено, защото кода е лесно обозрим и човек трудно ще сгреша при редакция. Въпреки това ние трябва да се абстрахираме от простотата на текущото решение и да гледаме на проблемите винаги в глобален мащаб. Независимо колко голяма или малка програма правим, трябва да я правим по най-добрия начин. Този начин на мислене трябва да ни стане навик, за да можем да пишем добре структуриран и лесно поддържан код.

За да решим задачата без повторение на циклите, можем да си зададем две променливи `min` и `max` и към тях да сложим стойностите на по-малкото и по-голямото число. След това лесно можем да напишем цикъл, който да върти от `min` до `max`, защото стойността на `min` винаги ще е по-малката, а на `max` – по-голямата.

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число a");
    int a = sc.nextInt();
    System.out.println("Въведете число b");
    int b = sc.nextInt();

    int min = a;
    int max = b;
}

```

```

        if(a > b){
            min = b;
            max = a;
        }

        for(int i = min; i <= max; i++){
            System.out.print(i + " ");
        }
    }

```

Така е много по-добре, защото нямаме повтаряне на код. И все пак можем да подобрим още програмата. Променливите `min` и `max` заемат памет, а можем да реализираме решението и без тях. Отново това може да ви се струва безсмислено, защото колко памет биха заемали две променливи .. (спомняте ли си ?). И все пак – въпросът е принципен. Винаги трябва да се стремим да пишем освен лесно поддържан и структуриран код, също така код, който е максимално опростен и лек. Този начин на мислене трябва да ни стане втора природа, за да можем да станем наистина добри програмисти.

Нека помислим как можем да се отървем от `min` и `max`. Лесно! Можем да си използваме променливите `a` и `b`, като изписваме цикъла винаги от `a` до `b`. Но ако `a` е по-голямо от `b`, просто можем да разменим стойностите на двете променливи и така `a` отново ще стане със стойност по-малка от `b`. По този начин цикълът от `a` до `b` винаги ще е правилен.

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число a");
    int a = sc.nextInt();
    System.out.println("Въведете число b");
    int b = sc.nextInt();

    if(a > b){//ако a > b, разменяме, за да може a
пак да стане по-малкото число.
        int temp = a;
        a = b;
        b= temp;
    }
}

```

```

        //цикълът винаги ще е от a до b.
        for(int i = a; i <= b; i++){
            System.out.print(i + " ");
        }
    }
}

```

Можем да си спестим дори и променливата **temp**, но това оставяме на вас.

5. Да се състави програма, която прочита две числа от конзолата и извежда сумата от техните факториели. Факториел на число N е резултата от умножението на всички числа от 1 до N .

Решение:

Нека разделим задачата на стъпки. Първата стъпка е да въведем две числа от конзолата. Това вече трябва да сме го затвърдили и да не е проблем да го реализираме. Следващата задача е да изчислим факториелите на двете числа. Както се споменава в условието, факториел се изчислява като се умножат всички числа от 1 до числото. Тоест ако имаме число N , трябва да умножим всички числа от 1 до N , за да намерим факториела му. Това може лесно да стане с цикъл, в който контролиращата ни променлива има начална стойност 1 и се увеличава с едно след всяка итерация, докато не стигне до стойност N . Преди цикъла ще инициализираме една променлива „**factorial**“, която ще умножаваме по всички цифри от 1 до N . В тялото на цикъла ще умножаваме контролиращата променлива с последната получена стойност от променливата **factorial**. След като знаем вече как да намерим факториела на едно число, е достатъчно да изведем факториелите на две въведени от конзолата числа, да ги съберем и получения резултат да покажем на екрана:

```

public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число a");
    int a = sc.nextInt();
    System.out.println("Въведете число b");
    int b = sc.nextInt();
}

```

```

        int factorialA = 1;
        for(int i = 1; i <= a; i++){
            factorialA *= i;
        }
        int factorialB = 1;
        for(int j = 1; j <= b; j++){
            factorialB *= j;
        }
        System.out.println(factorialA + factorialB);
    }

```

10. Да се състави програма, която приема число N в интервала [1..52]. Това число отговаря на номер на карта от наредено тесте с карти, като в тестето подредбата е следната:

Наредба по тежест на карта: 2,3,4,5,6,7,8,9,10, Вале, Дама, Поп, Асо.

Наредба по цвят на картите: спатия, каро, купа, пика.

Картите са подредени по следния начин: първо двойките от четирите бои, после тройките, после четворките и т.н.

Програмата да извежда картата, която съответства на въведения номер, след което да извежда списък с останалите карти от тестето, подредени от N до 51.

Пример:

Въведете стойност на x

47

Поп купа, Поп пика, Асо спатия, Асо каро, Асо купа, Асо пика

Решение:

След внимателен прочит на задачата стигаме до извода, че има зависимост между разположението на картите и номерацията им. Нека разгледаме по-внимателно: двойките са 1, 2, 3 и 4та карта. Тройките са 5,6,7 и 8та карта. Четворките са 9,10,11 и 12та карта и т.н. Тоест има зависимост между поредния номер на картата и силата на картата. Ако поредния номер на картата е X, то при X = 1, 2, 3 или 4, трябва да

покажем двойка. Ако $X = 5, 6, 7$ или 8 , трябва да покажем тройка и т.н. Можем да направим един switch, в който да имаме тринадесет случая – за всяка от тринадесетте степени на картите – от двойка до асо. $X = 1, 2, 3$ или 4 трябва да влезем в първи случай – двойки. При $X = 5, 6, 7$ или 8 – във втори случай и т.н. Спомняте си, че за да ползваме switch, ни трябва променлива, чиято стойност да проверяваме. Затова имаме нужда от стойност **cardNumber**, която е между 1 и 13. Каква е зависимостта между X и **cardNumber**? На пръв поглед **cardNumber** трябва да е равно на $X/4$ нали? Но при $X = 1, 2$ и 3 **cardNumber** излиза 0, а при $X = 4$ – **cardNumber** излиза 1, което не е вярно, защото първите четири поредни номера са карти с една и съща сила. Затова модифицираме формулата до $(x-1)/4+1$.

За да намерим коя боя е картата на пореден номер X , отново трябва да намерим зависимост. Ако имаме променлива **cardSuit**, каква е зависимостта спрямо X ? Нека погледнем кои карти са спатии – това са 1ва, 5та, 9та, 13та и т.н. Това са числа, при които остатъкът от делението на 4 е единица. Нека погледнем всички карти каро – това са 2-ра, 6та, 10та, 12та и т.н. - числа, чиито остатък от деление на 4 е равен на две. Аналогично намираме зависимост и за останалите две бои. Ето и кода:

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число от 1 до 52");
    int a = sc.nextInt();

    for(int card = a; card <= 52; card++) {
        int cardNumber = (card-1)/4+1;
        int cardSuit = card%4;
        switch(cardNumber){
            case 1: System.out.print("Двойка"); break;
            case 2: System.out.print("Тройка"); break;
            case 3: System.out.print("Четворка"); break;
            case 4: System.out.print("Петица"); break;
            case 5: System.out.print("Шестица"); break;
            case 6: System.out.print("Седмица"); break;
            case 7: System.out.print("Осмица"); break;
            case 8: System.out.print("Деветка"); break;
            case 9: System.out.print("Десятка"); break;
```



```

        case 10: System.out.print("Вале"); break;
        case 11: System.out.print("Дама"); break;
        case 12: System.out.print("Поп"); break;
        case 13: System.out.print("Асо"); break;
    }
    System.out.print(" ");
    switch(cardSuit) {
        case 1: System.out.print("спатия"); break;
        case 2: System.out.print("каро"); break;
        case 3: System.out.print("купа"); break;
        case 0: System.out.print("пика"); break;
    }
    if(card < 52){
        System.out.print(", ");
    }
}
}

```

Решения на задачи за едномерни масиви

1. Да се състави програма, която по въведен масив с четен размер да се конструира нов, като половината му елементи са точно като на оригиналния, а другите да са тези елементи, но в обратен ред. Последно, да се изведе новия масив на екрана.

Пример:

Въведете брой елементи:

6

Въведете елементите на масива:

4 2 5 3 1 2

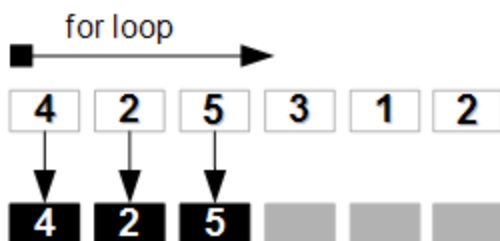
Новият масив:

4 2 5 2 1 3

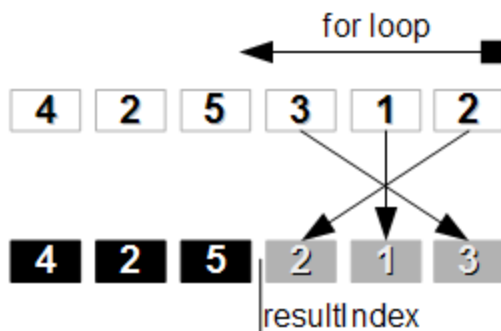
Решение:

Първото нещо, което трябва да направим, е да въведем елементите на масива от конзолата и да ги запишем в масив. За целта първо трябва да знаем каква е дължината на масива. След това трябва да инициализираме масив с такава дължина и в цикъл да заложим стойност на всяка клетка, като стойността ще вземем чрез скенер от конзолата.

В задачата се иска да конструираме нов масив. Ще го направим, като за дължина ще заложим дължината на нашия оригинален масив. Лесно можем да запълним половината от новия масив със стойностите от стария, като запишем цикъл, който върти индексите от 0 до половината от дължината на масива. В тялото на цикъла ще достъпваме клетките на новия масив и ще им подаваме стойността на съответната клетка от оригиналния масив.



Дотук добре. Но какво ще правим с втората половина от масива ? Стойностите трябва да са в обратен ред. Можем да направим втори цикъл, който да обхожда клетките на оригиналния масив отзад напред, докато не стигне до средата. Така ще обходим стойностите в ред, в който трябва да се подредят във втория ни масив. За да заложим стойностите на правилните места, обаче, във втория масив ние трябва да започнем да запълваме клетките от средата (защото тези преди средата вече са запълнени с първата половина на оригиналния масив). Можем да си инициализираме една променлива `resultIndex`, която да има стойност – средата на масива. Така, обхождайки отзад напред оригиналния масив, ние ще взимаме всяка стойност и ще я слагаме на клетка с индекс `resultIndex`, като след това ще инкрементираме променливата `resultIndex`, за да може следващата стойност да отиде в следващата клетка на нашия втори масив.



```

public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете брой елементи");
    int length = sc.nextInt();
    System.out.println("Въведете елементите на
масива");

    int[] arr = new int[length];
    //въвеждаме елементите на масива
    for(int i = 0; i < arr.length; i++) {
        arr[i] = sc.nextInt(); //всяка клетка се
запълва със стойност от конзолата
    }
    //създаваме втория масив
    int[] resultArray = new int[arr.length];
    //попълваме клетките на втория масив със
стойностите от първия, но само до половината.
    for(int i = 0; i < arr.length/2; i++) {
        resultArray[i] = arr[i];
    }
    //инициализираме помощна променлива, която ще
следи до коя клетка във втория масив сме стигнали
    int resultIndex = arr.length/2;
    //обхождаме първия масив от края до средата.

```

```

        for(int i = arr.length-1; i >= arr.length/2; i--)
    {
        resultArray[resultIndex] = arr[i]; //в
следващата поред клетка от втория масив слагаме стойността на
клетка от първия масив, като сме започнали отзад напред.
        resultIndex++; //увеличаваме стойността на
помощния индекс
    }
    //изписваме на екрана съдържанието на готовия
масив

    System.out.println("Новият масив:");
    for(int i = 0; i < resultArray.length; i++) {
        System.out.print(resultArray[i] + " ");
    }
}

```

5. Да се състави програма, чрез която се въвеждат цели числа в едномерен масив. Програмата да изчислява средната стойност от сумата на елементите, както и да изведе числото, което е най-близо до тази стойност.

Пример:

Въведете брой елементи:

7

Въведете елементите на масива:

1 2 3 4 5 6 7

Средна стойност:

4

Елемент най-близо до тази стойност:

4

Решение:

Лесно можем да изчислим средната стойност от сумата на елементите на масива. За целта трябва да обходим всички клетки чрез цикъл `for`, да съберем стойностите на клетките в отделна променлива **sum**, след което получения сбор да разделим на броя клетки (т.е. на дължината на масива). Оттук нататък, обаче, какво следва? Най-добрият начин да

изведем алгоритъм на една задача, е като се абстрахираме от програмистките термини и инструменти и се опитаме да я решим на лист хартия. Начертаваме си масива. Намираме средноаритметичното на всички елементи и .. какво правим ? Лесно виждаме кой е най-близкият елемент в масива. Но как става това ? Ние почти неволево пресмятаме, че най-близкия елемент е този, чиято разлика със средноаритметичната стойност е най-малка. Ами това е! Това е алгоритъма! Трябва да пресметнем разликата между всяка клетка на масива и средноаритметичното за целия масив. Резултатите от всички изваждания можем да запишем в нов масив. След това трябва да намерим индекса на елемента с най-малка стойност. Това ще е и индекса на елемента от оригиналния масив, в който се съдържа най-близкото число до средноаритметичното. И ето, че решихме задачата.



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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете брой елементи");
    int length = sc.nextInt();
    System.out.println("Въведете елементите на
масива");
```

```

int[] arr = new int[length];
for(int i = 0; i < arr.length; i++) {
    arr[i] = sc.nextInt();
}
//изчисляваме сумата от елементите
double sum = 0;
for(int i = 0; i < arr.length; i++) {
    sum+=arr[i];
}
//изчисляваме средноаритметичното от елементите
double avg = sum/arr.length;
System.out.println("Средна стойност: ");
System.out.println(avg);
//създаваме нов масив за разликите
double[] substractions = new double[arr.length];
for(int i = 0; i < substractions.length; i++) {
    double sub = arr[i] - avg;
    //премахваме знака от стойността
    double absSub = sub < 0 ? 0-sub : sub;
    //записваме стойността в помощния масив
    substractions[i] = absSub;
}
//намираме минималната стойност на помощния масив
и нейният индекс
double minSub = substractions[0];
int minIndex = 0;
for(int i = 1; i < substractions.length; i++) {
    if(minSub > substractions[i]) {
        minSub = substractions[i];
        minIndex = i;
    }
}
//индексът на минималната разлика е индексът на
елемента от оригиналния масив, най-близък до
средноаритметичното.
System.out.println("Елемент най-близък до тази
стойност:");

```

```
System.out.println(arr[minIndex]);
```

```
}
```

10. Да се състави програма, която по въведено число записва двоичното му представяне в едномерен масив. Да се изведе новополучения масив на екрана.

Пример:

Въведете число:

10

Новополученият масив е:

1 0 1 0

Решение:

Въвеждането на число вече ни е познато. Но как да разберем колко цифри съдържа двоичното му представяне? Нека си спомним как се трансформира десетично число в двоично ? Делим на две и записваме остатъка му, докато числото не стане нула. След това записваме всички остатъци отзад напред, като по този начин получаваме двоичното число. Точно този алгоритъм ще ползваме при решението на задачата. Понеже двоичното число трябва да изведем цифра по цифра в едномерен масив, за да инициализираме масива, ни трябва дължината му. Дължината можем да намерим, като ползваме част от алгоритъма за преобразуване на десетичното число в двоично – а именно, ще делим числото на две докато не получим нула, като ще преброим броя деления. Така ще намерим колко цифри ще има двоичното число. След като инициализираме масива със съответната дължина, остава да го обходим и да попълним остатъците от делението на числото на две. Важно е, обаче, да отбележим, че двоичното число се записва в ред, обратен на получаването на цифрите. Затова ще обходим масива отзад напред, за да разположим цифрите на двоичното число в правилния ред. Нека вземем за пример числото 41.

| | | |
|--------------------------------|---|------------------------------------|
| 41 : 2 = 20 с остатък 1 |  | Посока на записване на числото. |
| 20 : 2 = 10 с остатък 0 | | |
| 10 : 2 = 5 с остатък 0 | | |
| 5 : 2 = 2 с остатък 1 | | |
| 2 : 2 = 1 с остатък 0 | | |
| 1 : 2 = 0 с остатък 1 | | |

При деление на две, докато не стане нула, преброяваме 6 операции, затова ще инициализираме масив с 6 клетки. След това ще завъртим масива отзад напред, за да може първия положен остатък от делението на две да бъде записан в последната клетка на масива. По този начин числото ще се запише правилно в клетките на масива. Остава само да го изведем.

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

    Scanner sc = new Scanner(System.in);
    System.out.println("Въведете число");
    int a = sc.nextInt();
    int temp = a;

    int digits = 0;
    while(temp != 0) {
        temp = temp/2;
        digits++;
    }
    if(digits == 0) {
        digits = 1;
    }
    int[] array = new int[digits];
    for(int i = array.length-1; i>=0; i--){
        int digit = a%2;
        array[i] = digit;
    }
}
```



```
        a = a/2;
    }
    for(int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
```

Решения на задачи за двумерни масиви

1. Да се състави програма, която приема за входни данни двумерен масив и го отпечатва в редове и колони, като има празно място между всеки елемент от редовете.

Решение:

Можем да решим лесно задачата, като обходим индексите на всички редове от нулевия до дължината на масива и съответно всички колони от нулевата до дължината на съответния ред. Ще отпечатаме елементите разделени с интервал, като ще използваме операцията **System.out.print()** за не отиваме на нов ред. След края на вътрешния цикъл ще използваме **System.out.println()** без аргументи, за да отпечатаме нов ред на конзолата. Следва примерно решение:

MultiDimensionalArraysTask1.java

```
public class MultiDimensionalArraysTask1 {
    public static void main(String[] args) {
        // примерен масив
        int[][] array = {
            {3, 4, 1},
            {2, 1, 6},
            {0, 9, 7},
        }
```

```

        {8, -2, 5}

    };

    // за всеки ред от масива
    for (int row = 0; row < array.length; row++) {
        // за всяка колона в този ред
        for (int col = 0; col < array[row].length; col++) {
            // извеждаме поредния елемент на същия ред в конзолата
            System.out.print(array[row][col] + " ");
        }

        // след като сме извели всички елементи в този ред
        // извеждаме нов ред в конзолата
        System.out.println();
    }

}

```

5. Да се състави програма, която за подадена квадратна матрица $N \times N$ от числа намира произведението на елементите под главния диагонал.

Решение: Първо ще си нарисуваме един примерен двумерен масив с елементи, за да разгледам индексите на елементите, които ни интересуват:

| Индекси | 0 | 1 | 2 | 3 | 4 |
|---------|---|---|---|---|---|
| 0 | 1 | 5 | 2 | 3 | 3 |
| 1 | 6 | 2 | 4 | 5 | 9 |
| 2 | 4 | 3 | 2 | 4 | 2 |
| 3 | 7 | 5 | 1 | 4 | 1 |
| 4 | 2 | 1 | 4 | 2 | 7 |

Наблюдавайки индексите на елементите, лесно се забелязва че на 0-вия ред не вземаме елементите в произведението, на 1-вия – само един елемент, на 2-рия – два и т.н. Разсъждавайки по-нататък, виждаме, че на x -тия ред умножаваме първите x елемента. Можем да имплементираме решението инициализирайки променлива "product" със стойност 1 и след това да я умножаваме с всички елементи, които ни интересуват. Ще имаме два вложени цикъла – за редовете и за колоните, като ще обикаляме всичко редове, а колоните ще са от 0-вата до номера на предния ред. Следва решението на задачата:

MultiDimensionalArraysTask5.java

```
public class MultiDimensionalArraysTask5 {

    public static void main(String[] args) {
        // примерен масив
        int[][][] array = { { 3, 4, 1 },
                            { 2, 1, 6 },
                            { 8, 9, 7 },
                            { 8, 2, 5 }
                          };

        int product = 1;

        // за всеки ред от масива
        for (int row = 0; row < array.length; row++) {
```

```

        // за всяка колона в този ред до номера на
реда
        for (int col = 0; col < row; col++) {
            // умножаваме крайното произведение с
този елемент
            product *= array[row][col];
        }
        System.out.println("Резултатът е : " + product);
    }
}

```

10. Да се напише програма, която изисква от потребителя да въведе две числа М и N. След това да се построи матрица с размер М x N по следния начин (примерите са дадени за въведени N = 4, M = 5).

| | | | | |
|---|---|----|----|----|
| 1 | 8 | 9 | 16 | 17 |
| 2 | 7 | 10 | 15 | 18 |
| 3 | 6 | 11 | 14 | 19 |
| 4 | 5 | 12 | 13 | 20 |

Решение: Ще разделим задачата на няколко по-прости, които накрая ще модифицираме, за да получим финалното решение. Нека да попълним само първата колона като за начало. Ще си поддържаме едно число "counter" – текущото число, което ще записваме в поредната клетка. Ще обходим всички редове с един цикъл и в нулевата колоната ще записваме стойността на нашия брояч, след което ще я увеличаваме с единица. Можем лесно да реализираме това със следното парче код:

```

int arr[][] = new int[4][5];

int counter = 1;

```

```
// обикаляме редовете
for (int row = 0; row < arr.length; row++) {
    // и попълваме в нулевата колона текущото число
    // след което го увеличаваме
    arr[row][0] = counter;
    counter++;
}
```

Нека сега да попълним и втората колона. Решението е аналогично, с тази разлика, че попълваме от последния ред (с индекс `arr.length-1`) към първия (с индекс 0) и попълваме в колона с индекс 1.

```
// обикаляме редовете в обратен ред
for (int row = arr.length-1; row >= 0; row--) {
    // и попълваме в първата колона текущото число
    // след което го увеличаваме
    arr[row][1] = counter;
    counter++;
}
```

Сега лесно можем, обикаляйки всички колони да използваме или едното или другото запълване. Кога запълваме отгоре-надолу и отдолу-нагоре? Забелязваме, че колоните с четен индекс се запълват отгоре-надолу, а тези с нечетни – отдолу-нагоре. Така че, ще проверяваме на всяка стъпка за четността на колоната, която запълваме и ще използваме или единия или другия цикъл. Следва пълното решение на задачата:

MultiDimensionalArraysTask10.java

```
public class MultiDimensionalArraysTask10 {
    public static void main(String[] args) {
        int arr[][] = new int[4][5];

        int counter = 1;

        // обикаляме редовете
        for (int col = 0; col < arr[0].length; col++) {
```

```

        if (col % 2 == 0) {
            for (int row = 0; row < arr.length;
row++) {
                // и попълваме в нулевата колона
                // след което го увеличаваме
                arr[row][col] = counter;
                counter++;
            }
        } else {
            // обикаляме редовете в обратен ред
            for (int row = arr.length - 1; row >=
0; row--) {
                // и попълваме в първата колона
                // след което го увеличаваме
                arr[row][col] = counter;
                counter++;
            }
        }
    }
}

```

Решения на задачи за методи

1. Да се създаде метод, който приема за параметри масив от цели числа и връща като резултат средноаритметичното от стойностите в масива.

Решение: За да намерим средноаритметичното от елементите на масив като намерим тяхната сума и я разделим на броя елементи. Броят на елементите лесно се намира използвайки "arr.length". Как да намерим сумата? Ще си създадем една променлива, която ще инициализираме със стойност 0 и към нея ще добавяме всяко число. Ще обходим индексите на елементите от 0-вия до "arr.length" – 1, добавяме всеки

елемент и накрая ще върнем сумата разделена на броя. Можем да видим пълното решение по-долу:

MethodsTask1.java

```
public class MethodsTask1 {  
  
    static int avg(int[] arr) {  
        // тук ще трупаме сумата  
        int sum = 0;  
        // обхождаме всички индекси  
        for (int i = 0; i < arr.length; i++) {  
            // добавяме всеки елемент към сумата  
            sum += arr[i];  
        }  
        // връщаме сумата разделена на броя елементи  
        // за да намерим средноаритметичното  
        return sum / arr.length;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(avg(new int[]{3, 4, 1, 2, 5}));  
    }  
}
```

5. Да се състави метод, който приема за входни данни два масива от числа с равен брой. Методът да връща като резултат масив, съдържащ произведението на елементите от подадените два масива.

Решение: Обхождаме масивите като елемента от позиция "x" от единия масив умножаваме с елемента в същата позиция от другия масив и пак на същата позиция го записваме в новия масив. Предварително за всеки случай ще проверим дали масивите са с еднаква големина и също така ще заделим масив-резултат с големината на първия масив.

MethodsTask5.java

```

import java.util.Arrays;
public class MethodsTask5 {
    static int[] multArrays(int[] arr, int[] arr2) {
        // проверяваме дали масивите са с еднакъв размер
        if (arr.length == arr2.length) {
            // създаваме нов масив с големината на първия
            int result[] = new int[arr.length];
            // обикаляме всички елементи на масивите
            // и записваме произведението на елемент от
            // първия с елемент от втория
            for (int i=0; i<arr.length; i++) {
                result[i] = arr[i] * arr2[i];
            }
            return result;
        }
        return new int[0];
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(multArrays(new
int[] {3, 4, 1, 6, 4, 2, 8}, new int[] {3, 4, 1, 6, 4, 2, 8})));
    }
}

```

10. Да се създаде метод, който приема за параметри два масива от числа с произволен брой, но подредени в нарастващ ред. Методът да връща като резултат масив с дължина, равна на сбора от дължините на подадените масиви. Стойностите на масива трябва да са същите като стойностите на подадените масиви, но трябва да са подредени в нарастващ ред.

Решение: Ще си поддържаме два индекса за текущ елемент от първия и от втория масив. Сравняваме елементите на тези позиции и записваме на всяка стъпка по-малкия в масива-резултат. За тази цел там ще си пазим трети индекс – текуща позиция, на която ще записваме, след което ще я увеличаваме. От който от двата масива вземем елемент, там ще

придвижваме текущата позиция с единица. В един момент, вземайки ту от единия, ту от другия масив по-малкия елемент, някой от индексите ще излезе извън края на някой от масивите. В този момент цикъла ще спре, това ще е и нашето условие за изпълнение и след това от другия масив ще копираме останалите елементи в резултатния масив. Можем да видим примерно решение по-долу:

MethodsTask10.java

```
import java.util.Arrays;
public class MethodsTask10 {
    // метод, който по дадени 2 сортирани масива
    // ще връща нов, който съдържа елементите и на 2-та
    // и който отново ще е сортиран
    static int[] mergeArrays(int[] a, int[] b) {
        // нов масив с големина на сбора на подадените
        int[] result = new int[a.length + b.length];

        // индекси за текущите елементи, които ще
        // разглеждаме в първия и втория масив
        int indexA = 0;
        int indexB = 0;
        // до къде сме попълнили в новия масив
        int index = 0;

        // докато индексите ни в двата входни масива
        // са във валидни граници
        while (indexA < a.length && indexB < b.length) {
            // вземаме по-малкия от двата елемента
            // и го записваме на поредната позиция
            // в резултатния масив.
            // От който масив сме взели-увеличаваме
            // там текущия индекс на елемент с 1
            if (a[indexA] < b[indexB]) {
                result[index] = a[indexA];
                indexA++;
            } else {
                result[index] = b[indexB];
                indexB++;
            }
        }
    }
}
```

```

        // премества на следващия елемент
        // който ще попълваме в резултатния масив
        index++;
    }

    // ако са свършили елементите от втория масив
    // копираме останалите от първия
    // в резултатния масив
    while (indexA < a.length) {
        result[index] = a[indexA];
        indexA++;
        index++;
    }

    // ако са свършили елементите от първия масив
    // копираме останалите от втория
    // в резултатния масив
    while (indexB < b.length) {
        result[index] = b[indexB];
        indexB++;
        index++;
    }

    return result;
}

public static void main(String[] args) {
    System.out.println(Arrays.toString(
        mergeArrays(
            new int[] { 3, 5, 6, 7, 8, 10, 13 },
            new int[] { 1, 2, 2, 3, 4, 30, 40, 50 })));
}
}

```

Решения на задачи за рекурсия

1. Да се състави програма, която приема за входни данни естествено число N и изписва на екрана стойността на $N!$ (N факториел). Факториел на число N се изчислява чрез умножение на всички числа от 1 до N . Използвайте рекурсия.

Решение: За да решим една задача рекурсивно е необходимо да си дефинираме две неща – дъно на рекурсията и рекурентна зависимост. За дъно на рекурсията обособяваме базовите случаи, за които знаем решението на задачата – например $1! = 1$ и $0! = 1$. Така, че дъното ще бъде, че за дадено число $N \leq 1$, ще върнем 1 като резултат. Относно рекурентната зависимост трябва да се опитаме да сведем задачата до по-малка, дефинирана отново чрез себе си и някакви оператори. В случая можем да представим $N!$ като: $1 * 2 * 3 * 4 * \dots * N$. Можем да сведем задачата до по-малка, такава която се решава за по-малка стойност на N по следния начин :

$$N! = 1 * 2 * 3 * 4 \dots * N = 1 * 2 * 3 \dots * (N-1) * N = (N-1)! * N.$$

Т.е. използвайки, че предпоследния множител в това уравнение е $N-1$ можем да представим $N!$, чрез $(N-1)!$. С други думи, нашият метод, за дадено N , ще връща 1, при $N \leq 1$ и N умножено по резултата от нашия метод извикан за $N-1$. Можем да видим решението в кода по-долу:

RecursionTask1.java

```
public class RecursionTask1 {

    static long factorielOfN(int n) {
        // дъно на рекурсията
        if (n <= 1) {
            return 1;
        }
        // иначе умножаваме числото N
        // по (N-1)!
        return factorielOfN(n-1) * n;
    }

    public static void main(String[] args) {
        System.out.println("5! = " + factorielOfN(5));
    }
}
```

5. Да се състави програма, която приема за входни данни масив от цели числа. Програмата да извежда на екрана индекса на най-големия елемент. Използвайте рекурсия.

Решение: Ще си направим метод, който ще приема два параметъра – масива с числа и индекс на елемента от който нататък ще търсим в останалата част на масива, най-големия елемент. Нашата рекурсивна дефиниция ще бъде, че за даден индекс на елемент `index`, най-големия елемент е или на позиция `index`, или е на позиции от `index+1` до края на масива. Ако имаме един елемент, т.е. вече сме в края на масива, то връщаме индекса на този елемент – това ще бъде и дъното на нашата рекурсия. В реализацията на нашия рекурсивен метод, ще проверим първо дали не сме достигнали края на масива и ще върнем индекса на последния елемент, ако е така. В противен случай, ще извикаме рекурсивно този метод за същия масив, но да намери от текущата позиция нататък (`index+1`) най-големия елемент. Ще сравним текущия с намерения и ще върнем индекса на по-големия. Можем да видим решението по-долу:

RecursionTask5.java

```
public class RecursionTask5 {  
    // рекурсивен метод, който намира индекса на най-големия  
    // елемент в даден масив в интервала от index до  
    // дължината на масива - 1  
    // където index е параметър  
    static int indexOfLargestElement(int a[], int index) {  
        // дъно на рекурсията  
        // ако сме достигнали последния елемент  
        // връщаме неговия индекс  
        if (index == a.length - 1) {  
            return index;  
        }  
  
        // иначе намираме индекса на най-големия елемент в
```

```

        // останалата част на масива
        // на позиции от index+1 до a.length-1
        int maxElementIndex = indexOfLargestElement(a,
index + 1);

        // връщаме индекса на по-големия
        // или текущия или този, който сме намерили
        // в останалата част на масива
        if (a[index] > a[maxElementIndex]) {
            return index;
        } else {
            return maxElementIndex;
        }
    }

    public static void main(String[] args) {
        System.out.println(indexOfLargestElement(new int[]
{ 4, 1, 9, 4, 5, 1,-1, 0, 10, 4, 2, 1, 0 }, 0));
    }
}

```

10. Да се състави програма, която приема за входни данни две естествени числа – x и y . Програмата да пресмята стойността на x^y (x на степен y), като се използва единствено оператора събиране (+). Използвайте рекурсия.

Решение: За илюстрация на рекурсията, ще разработим два рекурсивни метода – единия ще умножава две числа рекурсивно, а другия – ще вдига рекурсивно число на дадена степен. Нека първо да разгледаме първия, вторият е аналогичен на него. Тъй като можем да използваме само събиране, умножението на числа можем да представим като сбор. Примерно : $3 * 5 = 3 + 3 + 3 + 3 + 3$. Можем да извлечем рекурсивна зависимост от този сбор по следния начин : $3*5 = 3 * 4 + 3$, т.е. да умножим с един път по-малко и да добавим самото число. Дъното на тази рекурентна зависимост е, когато искам да умножим едно число по 1 или 0. Тогава връщаме съответно самото число или числото 0. По аналогичен начин ще напишем и другия метод. Да вдигнем едно число

на някаква степен, можем да представим като да го вдигнем на степен с единица по-малка и след това да умножим по това число. Пример:

$$3^5 = 3 * 3 * 3 * 3 * 3 = 3^4 * 3$$

Т.е. отново можем да сведем задачата до по-малка на всяка стъпка, а използвайки първият рекурсивен метод можем да умножаваме. Базовият случай е, когато искаме да вдигнем число на 1-ва степен – тогава просто връщаме подаденото число. След примерен код на решението:

RecursionTask10.java

```
public class RecursionTask10 {

    // рекурсивен метод, който умножава едно число по друго
    // или с други думи - събира го със себе си, даден брой
    пъти
    static long multRecursively(long num, long times) {

        // проверки за дъно на рекурсията
        // 0 по всяко число е 0
        // и всяко число умножено по 1
        // дава себе си
        if (num == 0 || times == 0) {
            return 0;
        }
        if (times == 1) {
            return num;
        }

        // иначе събираме това число с резултата от
        // метода, но този път с един път по-малко
        return num + multRecursively(num, times - 1);
    }

    // рекурсивен метод, който вдига число на дадена степен
    static long powerRecursively(long num, long power) {
        // дъно на рекурсията
        // за степени 0 и 1 знаем отговора
    }
```

```

        if (power == 0) {
            return 1;
        }
        if (power == 1) {
            return num;
        }

        // в останалите случаи викаме метода, за да
пресметне
        // числото вдигнато на степен с 1 по-малка
        // и умножаваме числото по този резултат
        return multRecursively(powerRecursively(num,
power - 1), num);
    }

    public static void main(String[] args) {
        System.out.println(powerRecursively(7, 4));
    }
}

```

Решения на задачи за символни низове

1. Да се състави програма, чрез която се въвеждат два низа, съдържащи до 40 главни и малки букви. Като резултат на екрана да се извеждат низовете само с главни и само с малки букви.

Пример:

Въведете първи низ

aBcD

Въведете втори низ

eFGh

ABCD abcd

EFGH efgh

Решение: Ще използваме вградените операции на String – toUpperCase() и toLowerCase(), които съответно ще ни върнат низовете само с големи или само с малки букви. Ще използваме оператора "+", за да ги съберем добавяйки и по един интервал и така ще изведем резултата за първия и за втория низ. Можем да видим примерно решение по-долу:

```

StringsTask1.java

import java.util.Scanner;

public class StringsTask1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // въвеждаме двата низа
        System.out.println("Въведете първи низ");
        String s1 = sc.next();
        System.out.println("Въведете втори низ");
        String s2 = sc.next();

        // изкарваме ги като използваме операциите на String
        // за преобразуване на низовете само с малки или големи
        букви
        System.out.println(s1.toUpperCase() + " " +
s1.toLowerCase());
        System.out.println(s2.toUpperCase() + " " +
s2.toLowerCase());
        sc.close();
    }
}

```

5. Да се състави програма, която приема за входни данни дума. Програмата да извежда като резултат друга дума, като буквите ѝ са получени като към всеки код на буква е добавено числото 5 и е записана новополучената буква.

Пример:

Въведете дума

Hello

Mjqqt

Решение: Можем да вземем знак от даден низ използвайки операцията `charAt()`, която по даден индекс ще ни върне символът на този индекс. Обхождаме с един цикъл всички валидни индекси на символи от нулевия до дължината на низа минус едно. След като вземем поредния символ конструираме нова променлива от тип `char`, като добавяме числото 5 към този символ и тъй като резултата от тази операция ще бъде цяло число от тип `int` - преобразуваме този израз с конструкцията `"(char)"` отново към променлива от знаков тип. Накрая извеждаме новият символ на същия ред в конзолата. След като цикъла завърши, ще поставим нов ред и ще затворим стандартния вход. Следва примерно решение:

StringsTask5.java

```
import java.util.Scanner;

public class StringsTask5 {
    public static void main(String[] args) {
        System.out.println("Въведете дума");
        Scanner sc = new Scanner(System.in);

        // въвеждаме дума от конзолата
        String word = sc.next();

        // обхождаме всички знаци думата
        for (int letterIndex=0; letterIndex <
word.length(); letterIndex++) {
            // конструираме нов знак
            // като вземаме знака на текуща позиция от
низа

            // и добавяме числото 5
            // конвертираме резултата към тип char, тъй
```

```

като резултата
        // от това събиране е от тип int
        char newLetter = (char)
(word.charAt(letterIndex) + 5);

        // извеждаме на същия ред поредната нова
буква
        System.out.print(newLetter);
    }

    System.out.println();
    sc.close();
}
}

```

10. Да се състави програма, която по въведен масив от символни низове отпечатва тези низове в рамка. Рамката да е съставена от символа *.

Пример:

```

Въведете дължина на масива
5
Въведете дума
Hello
Въведете дума
World
Въведете дума
in
Въведете дума
a
Въведете дума
frame
*****
*Hello*
*World*
*in   *
*a    *
*frame*
*****

```

Решение: Ако си разпишем няколко примера с произволни думи, ще забележим, че дължината на рамката е равна на дължината на най-дългата дума плюс 2, заради двете вертикални линии в двата края на рамката. Така, че след като прочетем всички думи, ще намерим дължината на най-дългата дума. Преди да изведем думите, заедно с вертикалните рамки ще изведем толкова на брой плюс 2 звездички. Същото ще направим и накрая, за да затворим рамката. По-нататък ще обходим всички думи, първо ще изведем звездичка (за лявата вертикална рамка), след това поредната дума, и след това толкова на брой интервали, колкото е разликата между най-дългата дума и текущата. Правим това за да бъдат звездичките от дясната вертикална рамка добре подравнени и да оформят една линия. Накрая ще изведем още една звездичка за дясната рамка и ще отидем на нов ред. За по-добра четимост, ще разделим имплементацията на няколко метода – за четене на входните данни, отпечатване на хоризонтална линия от звезди, намиране на дължината на най-дългата дума, отпечатване на определен брой интервали, отпечатване на думите, заедно с вертикалните линии от звезди и главен метод, който ще извиква останалите. Следва примерно решение на задачата:

```
StringsTask10.java

import java.util.Scanner;

public class StringsTask10 {
    // метод, който ще прочита броя думи
    // както и самите думи от конзолата
    // като резултат ще връща масив от низове
    static String[] readWords() {
        System.out.println("Въведете дължина на масива");
        Scanner sc = new Scanner(System.in);
        int wordsCount = sc.nextInt();
        String[] words = new String[wordsCount];

        for (int wordIndex=0; wordIndex < words.length;
wordIndex++) {
            System.out.println("Въведете дума");
```

```

        words[wordIndex] = sc.next();
    }
    sc.close();
    return words;
}

// отпечатва хоризонтална линия от
// звездички по дадена дължина
static void printBorder(int length) {
    for (int index=0; index<length; index++) {
        System.out.print("*");
    }
    System.out.println();
}

// връща дължината на най-дългия низ
// по даден масив от низове
static int findMaxLength(String[] words) {
    int maxLength = 0;
    for (int wordIndex=0; wordIndex < words.length;
wordIndex++) {
        if (words[wordIndex].length() > maxLength) {
            maxLength = words[wordIndex].length();
        }
    }
    return maxLength;
}

// отпечатва определен брой интервали
// по зададен параметър
static void printSpaces(int numOfSpaces) {
    for (int spaces=0; spaces < numOfSpaces; spaces++) {
        System.out.print(" ");
    }
}

// отпечатва думите, като им слага вертикални рамки
// в краищата
static void printWords(String[] words, int maxLength) {
    // обхождаме всички думи по индекс
    for (int wordIndex=0; wordIndex < words.length;
wordIndex++) {
        // слагаме рамка в началото

```

```

        System.out.print("*");

        // отпечатваме текущата дума
        System.out.print(words[wordIndex]);
        // отпечатваме толкова на брой интервали
        // колкото е разликата между най-дългата
дума и текущата
        // за да бъдат подравнени звездичките
        printSpaces(maxLength-
words[wordIndex].length());

        // слагаме "*" за крайната рамка
        System.out.print("*");

        System.out.println();
    }
}
// главен метод, в който извикваме останалите
public static void main(String[] args) {
    String[] words = readWords();
    int maxLength = findMaxLength(words);
    printBorder(maxLength+2);
    printWords(words, maxLength);
    printBorder(maxLength+2);
}
}

```

Решения на задачи за сортиращи алгоритми и двоично търсене

1. Да се състави програма, която се опитва на всяка стъпка да познае намислено от потребителя число в интервала от 0 до 100. На всяка стъпка програмата извежда предположение за намисленото от потребителя число и като отговор прочита цяло число – 0, 1 или 2. Като съответно 0 означава – надолу, 2 – нагоре, а 1 – числото е познато. Програмата трябва познае числото за не повече от 8 питания, след което да приключи работа.

Пример:

```
30?
2
40?
2
70?
0
60?
1
```

Решение: Ще решим задачата използвайки двоично търсене. Ще си поддържаме две променливи – left и right, които ще ни показват началото и края на интервала, в който ще се намира числото, което потребителя си е намислил. Ще намерим средата на интервала, която е $(\text{left} + \text{right}) / 2$ и всеки път ще питаме за това число. Ако средата е по-малка ще модифицираме левия край да почва от средата, в противен случай – десния. Ако пък сме познали числото, то ще прекъснем цикъла, в който задаваме въпроси към потребителя. Можем да видим примерно решение по-долу:

SortingTask1.java

```
import java.util.Scanner;

public class SortingTask1 {
    public static void main(String[] args) {
        // начало на интервала, в който ще търсим
        int left = 0;
        // край на интервала, в който ще търсим
        int right = 100;
        Scanner sc = new Scanner(System.in);
        // докато нашия интервал е валиден
        while (left <= right) {
            // намираме текущата среда на интервала
            int middle = (left + right) / 2;

            // питаме потребителя и прочитаме отговора
            System.out.println(middle + "?");
```

```

        int answer = sc.nextInt();

        // ако отговора е "надолу"
        // модифицираме десния край да е в ляво
        // от средата
        if (answer == 0) {
            right = middle-1;
        }

        // ако сме го познали прекъсваме
        if (answer == 1) {
            break;
        }
        // ако отговора е "нагоре"
        // модифицираме левия край да е в дясно
        // от средата
        if (answer == 2) {
            left = middle+1;
        }
    }
}

```

5. Напишете програма, която по дадена последователност от нули и единици ги изкарва сортирани. Да се реализира по възможно най-оптималния начин.

Решение: Най-ефективно можем да реализираме това използвайки модификация на сортиране чрез броене. Можем да го опростим още, като вместо масив си пазим две променливи, които да ни пазят броя единици и нули. Накрая ще изведем толкова на брой нули и единици, колкото сме преброили в двете променливи. Можем да намалим допълнително сложността по памет, като не пазим входния масив, а докато го четем елемент по елемент да преброяваме 1-ците и 0-те в него. Можем да видим примерен код по-долу:

SortingTask5.java

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

```

public class SortingTask5 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(); // брой елементи
        int zeros = 0; // брой нули
        int ones = 0; // брой единици

        for (int index=0; index<n; index++) {
            // прочитаме поредния елемент
            int element = sc.nextInt();
            if (element == 0) {
                // ако е 0 - преброяваме я
                zeros++;
            }
            if (element == 1) {
                // ако е 1-преброяваме в друга променлива
                ones++;
            }
        }

        for (int i=0; i < zeros; i++) {
            // извеждаме първо нулите
            System.out.print(0 + " ");
        }
        for (int i=0; i < ones; i++) {
            // а след това и единиците
            System.out.print(1 + " ");
        }
        System.out.println();
    }
}

```

10. Да се реализира структура от данни пирамида и да се приложи към алгоритъма за сортиране чрез пряка селекция, като на всяка стъпка най-големия елемент да се вади от пирамидата и да се слага в края на нов масив. Да се изведе новополучения сортиран масив на екрана.

Решение: Ще използваме директно имплементацията на пирамидална сортировка, както е показана в глава 13-та. Ще я модифицираме, така че най-големия елемент след като се размени с последния, да се записва след това на всяка стъпка в нов масив. Ще си поддържаме една променлива за броя елементи в този масив, като ще записваме нов елемент и накрая ще увеличаваме променливата. Правим това, за да знаме на кой индекс да запишем всяка поредна стойност. Накрая ще изведем получения масив. Следва примерен код:

```

SortingTask10.java

public class SortingTask10 {
    private static int N;//брой елементи в масива

    /* Начало на програмата */
    public static void main(String[] args) {
        int[] arr = {9,13,11,4,28,75,5,26,1};
        arr = sort(arr);

        System.out.println("\nElements after sorting ");
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    /* Метод за сортиране */
    public static int[] sort(int arr[]) {
        int result[] = new int[arr.length]; // масив, в
        който ще слагаме елементите един по един
        int count = 0; // брой извадени елементи
        buildMaxHeap(arr);

        for (int i = N; i > 0; i--) {
            result[count] = arr[0]; // вземаме най-
            големия от пирамидата
            count++;

            swap(arr, 0, i);//разменяме първия елемент
            с последния
            N = N - 1;//намаляваме N, за да може

```

пирамидата да не взема предвид извадения елемент от края на масива

`upHeap(arr, 0);` // пренаареждане на остатъка от масива до изпълнение на пирамидалното условие

```

    }
    return result;
}

/* Метод за "пирамидизиране" */
public static void buildMaxHeap(int arr[]) {
    N = arr.length - 1;
    for (int i = N / 2; i >= 0; i--)
        upHeap(arr, i);
}

public static void upHeap(int arr[], int i) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int max = i;
    if (left <= N && arr[left] > arr[i])
        max = left;
    if (right <= N && arr[right] > arr[max])
        max = right;

    if (max != i) {
        swap(arr, i, max);
        upHeap(arr, max);
    }
}

/* Метод, който разменя два елемента */
public static void swap(int arr[], int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
}

```