Д И П Л О М Н А Р А Б О Т А

На тема:

Учебно помагало „Въведение в програмирането с Java“

Дипломант:Христо Александров Хаджиценев

Фак.номер: 0701181049

Научен ръководител: доц. Д-р Коста Гъров

Съдържание:

1. Увод

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

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

Започвайки с програмирането от езика и платформата Java вие поемате по един много перспективен път. Ако отворите някой сайт с обяви за работа за програмисти, ще се убедите, че търсенето на Java специалисти е огромно и е по-голямо в сравнение с всякакви други езици и платформи (примерно C#, PHP или C++).

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

2. Въведение в програмирането.

Същност на програмирането

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

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

В този учебник ще разгледаме програмния език Java. Това е език за програмиране от високо ниво.

Нашата първа Java програма

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

```
classHelloJava{
        publicstaticvoidmain(String[]arguments){
                System.out.println("Hello, Java");
        }
}
```

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

Как работи нашата първа Java програма?

Нашата първа програма е съставена от три логически части:

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

Дефиниция на клас

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

Дефиниция на метод main()

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

```
publicstaticvoidmain(String[]arguments)
```

Методът трябва да е деклариран по точно показания начин, трябва да е **public**, **static**и **void**, трябва да има име **main** и като списък от параметри трябва да има

един единствен параметър от тип масив от **String**. Местата на модификаторите**public**и **static**могат да се разменят. В нашия пример параметърът се казва **arguments**, но това не е задължително, параметърът може да има произволно име. Повечето програмисти избират за име **args** или **argv**.

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

Съдържание на main() метода

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

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

Java различава главни от малки букви!

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

	**Внимавайте, докато пишете! Изписването на един и същ текст с главни, малки букви или смесено в Java означава различни неща. Да напишем Classe различно отclassи да напишемSystem.oute различно от SYSTEM.OUT.**

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

Програмният код трябва да е правилно форматиран

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

```
classHelloJava{
        publicstaticvoidmain(String[]arguments){
                System.out.println("Hello, Java");
        }
}
```

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

```
classHelloJava{
publicstaticvoidmain(String[]arguments){
System.out.println("Hello, Java");
}
}
```

или на един ред:

```
classHelloJava{publicstaticvoid                main(String[]arguments){
System.out.println("Hello, Java");}}
```

или дори така:

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

```
}
```

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

Имената на файловете съответстват на класовете

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

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

Езикът Java

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

Ключови думи

Езикът Java използва следните ключови думи:

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

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

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

Автоматично управление на паметта

Едно от най-големите предимства на Java е предлаганото от нея автоматично управление на паметта. То предпазва програмистите от сложната задача сами да заделят памет за обектите и да следят подходящия момент за нейното освобождаване. Това рязко засилва производителността на програмистите и увеличава качеството на програмите, писани на Java.

За управлението на паметта се грижи специален компонент от виртуалната машина, наречен галено "**събирач на боклука**" или "**система за почистване на**

паметта" (**Garbage Collector**). Основните задачи на събирача на боклука са да следи кога заделената памет за променливи и обекти вече не се използва, да освобождава тази памет и да я прави достъпна за последващи заделяния.

3. Примитивни типове данни.

Какво е променлива?

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

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

Типове данни

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

Характеристики

Типовете данни се характеризират с:

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

Видове

Типовете данни се разделят на следните видове:

- Целочислени типове – **byte**, **short**, **int**, **long**;

- Реални типове с плаваща запетая – **float** и **double**;
- Булев тип –**boolean**;
- Символен тип – **char**;
- Обектен тип – **Object**;
- Символни низове – **String**.

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

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

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

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

Целочислени типове

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

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

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

Следващият целочислен тип, който ще разгледаме е типът **int**. Той е 32- битов целочислен знаков тип. Както виждаме, с нарастването на битовете нарастват и възможните стойности, които даден тип може да заема. Стойността по подразбиране е числото 0. Минималната стойност, която заема е -2147483648, а максималната +2147483647.

Последният целочислен тип, който ни предстои да разгледаме, е типът **long**. Той е 64-битово цяло число със знак със стойност по подразбиране **0L**. Знакът **L** се указва, че числото е от тип long (иначе се подразбира **int**). Минималната стойност, която типът **long** заема, е -9223372036854775808, а максималната +9223372036854775807.

Целочислени типове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни целочислени типове, инициализираме ги и ги отпечатваме на конзолата.

```
byte centuries = 20;
short years = 2000;
int days = 730480;
long hours = 17531520;
System.out.println(centuries + " centuries is " + years +
" years, or " + days + " days, or " + hours + " hours.");
```

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

Нека видим резултата от горния фрагмент код отдолу.

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

Реални типове с плаваща запетая

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

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

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

Реални типове – пример

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

```
float floatPI = 3.14f;
double doublePI = 3.14;
```

Булев тип

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

Булев тип – пример

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

```
int a = 1;
int b = 2;
boolean greaterAB = (a > b);
boolean equalA1 = (a == 1);
if (greaterAB) {
        System.out.println("A > B");
} else {
        System.out.println("A <= B");
}
System.out.println("greaterAB = " + greaterAB);
System.out.println("equalA1 = " + equalA1);
```

Нека видим резултата от горния фрагмент код:

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

В примера декларираме две променливи от тип **int**, сравняваме ги и резултата го присвояваме на променливата от булев тип **greaterAB**. Аналогично за

променливата **equalA1**. Ако променливата**greaterAB** е **true**, на конзолата се отпечатва **A>B**, в противен случай **B>A**.

Символен тип

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

Символен тип – пример

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

```
char symbol = 'a';
System.out.println(
"The code of '" + symbol + "' is: " + (int) symbol);
symbol = 'b';
System.out.println(
"The code of '" + symbol + "' is: " + (int) symbol);
symbol = 'A';
System.out.println(
"The code of '" + symbol + "' is: " + (int) symbol);
```

Нека видим резултата от горния фрагмент код отдолу:

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

Символни низове (стрингове)

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

Символни низове – пример

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

```
String firstName = "Ivan";
String lastName = "Ivanov";
String fullName = firstName + " " + lastName;
System.out.println("Hello, " + firstName + "!");
System.out.println("Your full name is " +
fullName + ".");
```

Да видим резултата от горния фрагмент код:

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

Обектен тип

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

Използване на обекти – пример

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

```
Object container = 5;
Object container2 = "Five";
System.out.println("The value of container is: " + container);
System.out.println("The value of container2 is: " + container2);
```

Нека видим резултата от горния фрагмент код:

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

4. Променливи.

Какво е променлива?

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

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

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

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

Характеристики на променливите

Променливите се характеризират с:

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

- Стойност (запазената информация).

Деклариране на променливи

Когато декларираме променлива, ние:

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

Ето какъв е синтаксисът за деклариране на променливи:

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

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

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

Присвояване на стойност

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

Присвояване на стойност – примери

Ето няколко примера за присвояване на стойност на променлива:

```
int firstValue = 5;
int secondValue;
```

```
int thirdValue;
secondValue = firstValue;
thirdValue = firstValue = 3;
```

Стойностни и референтни типове

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

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

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

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

Упражнения

1. Декларирайте няколко променливи, като изберете за всяка една най-подходящия от типовете **byte**, **short**, **int** и **long**, за да представят следните стойности: 52130; -115; 4825932; 97; -10000.
2. Кои от следните стойности може да се присвоят на променлива от тип **float** и кои на променлива от тип **double**: 34.567839023; 12.345; 8923.1234857; 3456.091?
3. Инициализирайте променлива от тип **int** със стойност 256 в шестнадесетичен формат (256 е 100 в бройна система с база 16).
4. Декларирайте променлива **isMale** от тип **boolean**и присвоете стойност на последната в зависимост от вашия пол.
5. Декларирайте две променливи от тип **String** със стойности "Hello" и "World". Декларирайте променлива от тип **Object**. Присвоете на тази променлива стойността, която се получава от конкатенацията на двете стрингови променливи (добавете интервал, ако е необходимо). Отпечатайте променливата от тип **Object**. Декларирайте променлива от тип **String** и присвоете на последната променливата от тип **Object**.

5. Оператори и изрази.

Оператори

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

Какво е оператор?

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

Операторите в Java

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

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

- Оператори за преобразуване на типовете – позволяват преобразуването на данни от един тип в друг.

Категории оператори

Следва списък с операторите, разделени по категории:

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

Оператори според броя аргументи

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

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

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

Някой оператори в Java извършват различни операции, когато се приложат с различен тип данни. Пример за това е операторът +. Когато се използва с числени типове данни (**int**, **long**, **float** и др.), операторът извършва операцията математическо събиране. Когато обаче използваме оператора със символни

низове, той слепва съдържанието на двете променливи / литерали и връща новополучения низ.

Оператори – пример
Ето един пример за използване на оператори:

```
int z = 4 + 8;
System.out.println(z); // 12
String firstName = "Lachezar";
String lastName = "Bozhkov";
String fullName = firstName + " " + lastName;
System.out.println(fullName);
```

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

Аритметични оператори

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

Когато използваме операторите **++** и **--** като суфикс (поставяме операторът непосредствено пред променливата) първо се пресмята новата стойност, а после се връща резултата и програмата продължава с решението на израза, докато при използването на операторите като постфикс (поставяме оператора

непосредствено след променливата) първо се връща оригиналната стойност на операнда, а после се добавя или изважда единица към нея.

Аритметични оператори – примери

Ето няколко примера за аритметични оператори:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
System.out.println(squareSide); // 4.25
System.out.println(squareArea); // 18.0625
int a = 5;
int b = 4;
System.out.println(a + b);      // 9
System.out.println(a + b++);    // 9
System.out.println(a + b);      // 10
System.out.println(a + (++b));  // 11
System.out.println(a + b);      // 11
System.out.println(14 / a);     // 2
System.out.println(14 % a);     // 4
```

Логически оператори

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

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

x	y	!x	x && y	x || y	x ^ y

true	**true**	**false**	**true**	**true**	**false**
true	**false**	**false**	**false**	**true**	**true**
false	**true**	**true**	**false**	**true**	**true**
false	**false**	**true**	**false**	**false**	**false**

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

Логически оператори – пример

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

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

Оператор за съединяване на низове

Оператора + се използва за съединяване на символни низове (**String**). Това, което прави операторът е просто слепя два или повече низа и връща резултата

като един нов низ. Ако поне един от аргументите в израза е от тип **String**, и има други операнди, които не са от тип **String**, то те автоматично ще бъдат преобразувана към тип **String**.

Оператор за съединяване на низове – пример

Ето един пример, в който съединяваме няколко символни низа:

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

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

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

Побитови оператори

Побитов оператор означава оператор, който действа над двоичното представяне на числовите типове. В компютрите всички данни и в частност числовите данни

се представят като поредица от нули и единици. За целта се използва **двоичната бройна система**. Например числото **55** в двоична бройна система се представя като **00110111.**

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

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

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

Побитови оператори – пример

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

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

System.out.println( a | b); // 0000 0111 = 7
System.out.println( a & b); // 0000 0001 = 1
System.out.println( a ^ b); // 0000 0110 = 6
```

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

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

Оператори за сравнение

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

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

Всички оператори за сравнение са двуаргументни (приемат два операнда), а върнатият от тях резултат е булев (**true** или **false**). Операторите за сравнение

имат по-малък приоритет от аритметичните, но са с по-голям от операторите за присвояване на стойност.

Оператори за сравнение – пример

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

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

В примерната програма, първо създадохме двете променливи **x** и **y** и им присвоихме стойностите 10 и 5. На следващия ред отпечатваме на конзолата, посредством метода **println()** на **System.out**, резултата от сравняването на двете променливи **x** и **y** посредством оператора >. Върнатият резултат е **true**, защото **x** има по-голяма стойност от **y**. На следващите 5 реда се отпечатва върнатият резултат от използването на останалите 5 оператора за сравнение с променливите **x** и **y.**

Оператори за присвояване

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

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

Оператори за присвояване – пример

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

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

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

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

Условен оператор

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

Синтаксисът на оператора е следният:

```
операнд1 ? операнд2 : операнд3
```

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

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

Условен оператор – пример

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

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

Други оператори

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

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

Изрази

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

```
int r = (150-20) / 2 + 5;
double surface = Math.PI * r * r;
double perimeter = 2 * Math.PI * r;
System.out.println(r);
System.out.println(surface);
System.out.println(perimeter);
```

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

```
70
15393.804002589986
439.822971502571
```

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

```
int a = 5;
int b = ++a;
System.out.println(a); // 6
System.out.println(b); // 6
```

Упражнения:

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

6. Вход и изход от конзолата.

Какво представлява конзолата?

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

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

Комуникация между потребителя и програмата

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

споменахме, едно от средствата за комуникация между програмите и потребителят е конзолата, но тя става все по-рядко използвана. Това е така, понеже съвременните средства за комуникация са по-удобни и интуитивни за работа.

Кога да използваме конзолата?

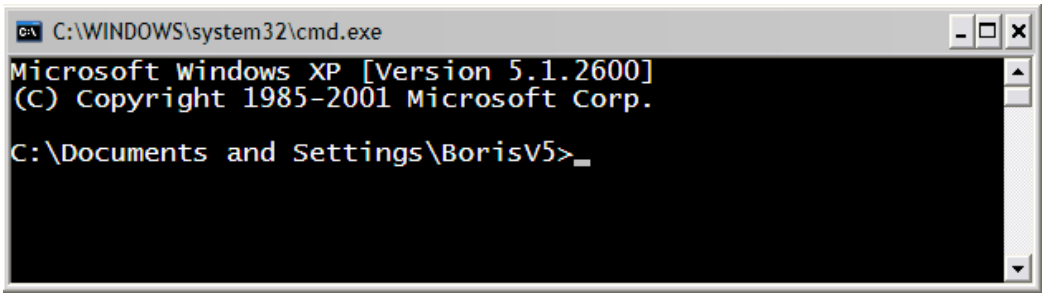
В някои случаи, конзолата си остава незаменимо средство за комуникация. Един от тези случаи е писане на малки и прости програмки, където по-важното е вниманието да е насочено към конкретния проблем, който решаваме, а не към елегантно представяне на резултата на потребителя. Тогава се използва просто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

Как да стартираме конзолата?

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

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

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



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

Подробно за конзолите

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

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

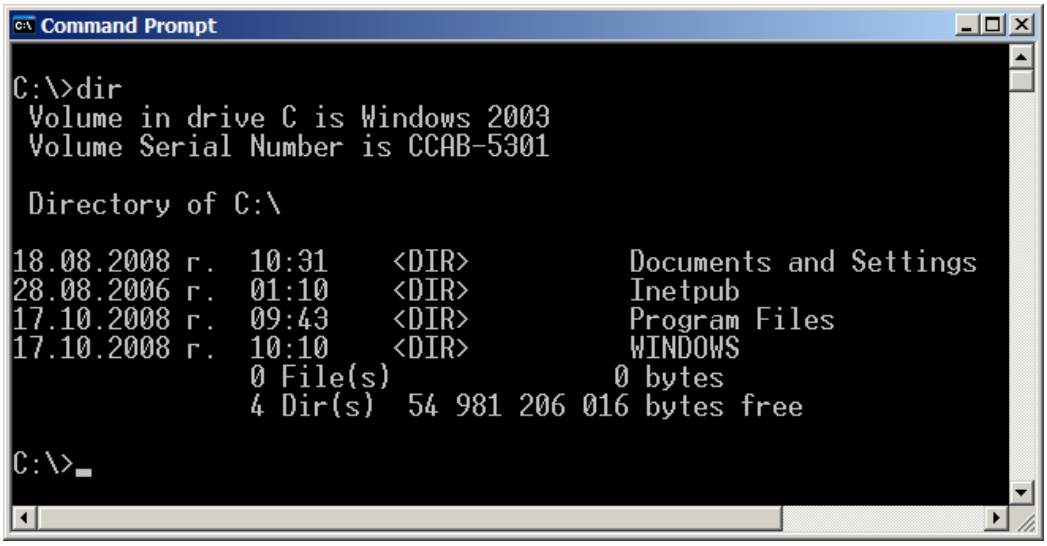
Така наречените "обвивки", могат да се разгледат в две основни категории, според това какъв интерфейс могат да предоставят към операционната система.

- Команден интерфейс (CLI – Command Line Interface) – представлява конзола за команди (като например **cmd.exe**).
- Графичен интерфейс (GUI – Graphical User Interface) – представлява графична среда за работа (като например Windows Explorer).

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

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

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



Основни конзолни команди

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

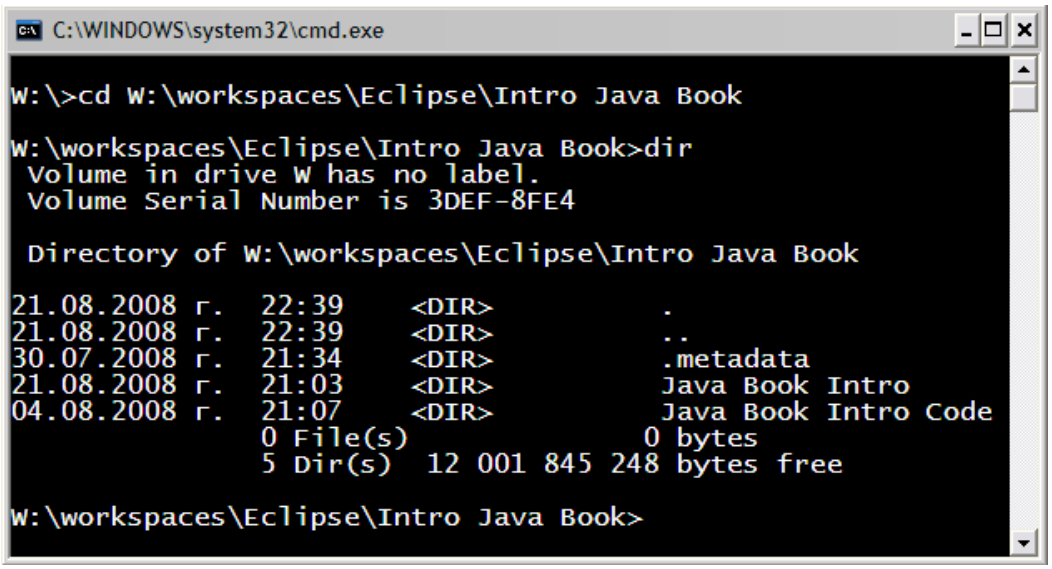
Конзолни команди под Windows

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

Команда	**Описание**
dir	Показва съдържанието на текущата директория.
cd <directory name>	Променя текущата директория.
mkdir <directory name>	Създава нова директория в текущата.
rmdir <directory name>	Изтрива съществуваща директория.

type <file name>	Визуализираща съдържанието на файл.
copy <src file><destination file>	Копира един файл в друг файл.

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



Стандартен вход-изход

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

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

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

Печатане на конзолата

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

Стандартен вход и стандартен изход

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

Концепцията за потоците ще бъде подробно разгледана по-късно. Засега ще се съсредоточим върху теоретичната основа, засягаща програмния вход и изход в Java.

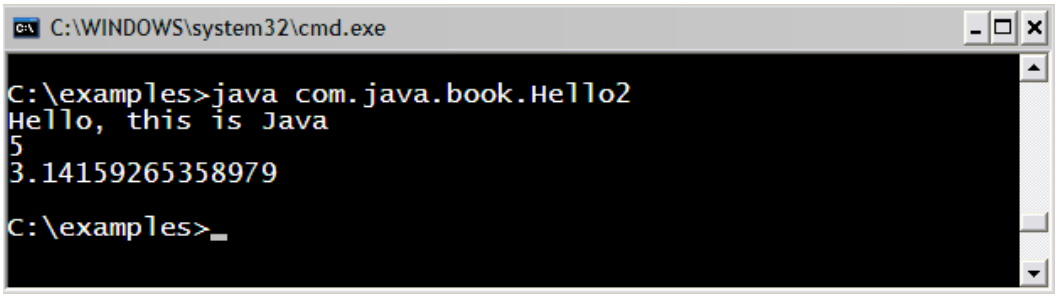
Използване на print() и println()

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

Ето някой примери за отпечатването на различни типове данни:

```
System.out.println("Hello, this is Java");
System.out.println(5);
System.out.println(3.14159265358979);
```

Резултатът от изпълнението на този код изглежда така:



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

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

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

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

Изходът от този пример е:

```
I love
this Book!
```

Забелязваме, че изхода от примера е отпечатан на два реда, независимо че кодът е на три. Това е така, понеже на първия ред от кода използваме **println(),** по този начин се отпечатва "**I love**" и след това се минава на нов ред. В следващите два

реда от кода се използва метода **print,** които печата, без да минава на нов ред и по този начин думите "**this**" и "**Book!**" си остават на един ред.

Форматиран изход с printf()

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

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

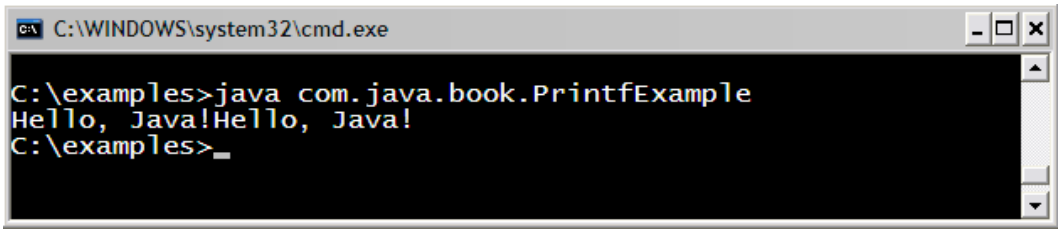
```
printf(<formatted string>, <param1>, <param2>, <param3>, …)
```

Форматиран изход с printf() – примери

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

```
String str = "Hello, Java!";
System.out.print(str);
System.out.printf("%s", str);
```

Резултатът от изпълнението на този пример е:



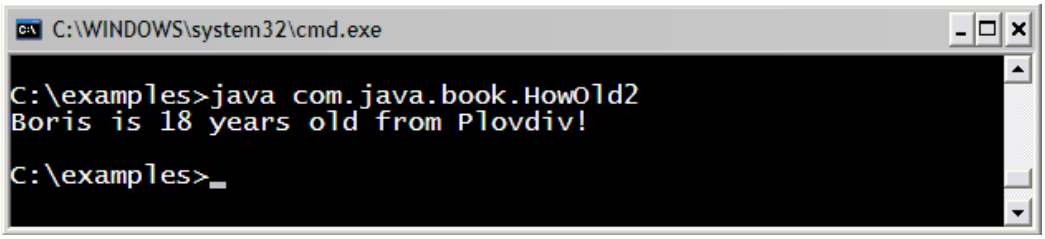
Виждаме като резултат, два пъти "**Hello, Java!**" на един ред. Това е така, понеже никъде в програмата нямаме команда за нов ред.

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

Следващият пример ще разясни допълнително концепцията:

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

Резултатът от изпълнението на примера е:



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

Вход от конзолата

Както в началото на темата обяснихме, най-подходяща за малки приложения е конзолната комуникация, понеже е най-лесна за имплементиране. **Стандартното входно устройство** е тази част от операционната система, която контролира от къде програмата ще получи своя вход. По подразбиране "стандартното входно устройство" чете своя вход от драйвер "закачен" за клавиатурата. Това може да бъде променено и стандартният вход може да бъде пренасочен към друго място, например към файл.

Всеки език за програмиране има механизъм за четене и писане в конзолата. Обектът, контролиращ стандартния входен поток в **Java,** е **System.in**.

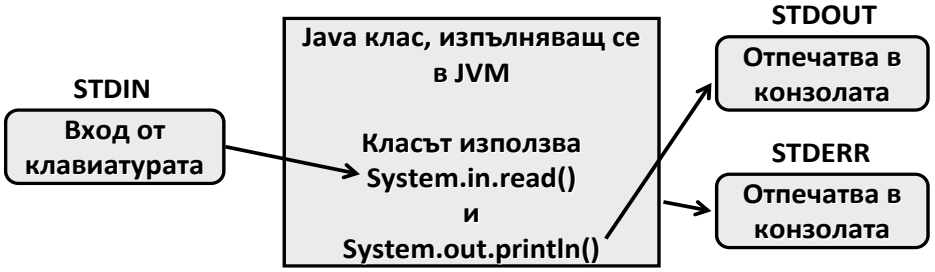
От конзолата можем да четем различни данни:

- текст;
- числени типове, след конвертиране.

Четене от потока System.in

Да си припомним, че системният клас **System** има статична член-променлива **in**. Тази член променлива е от тип **InputStream**. Това е базов клас (предшественик) на всички класове, представящи входен поток от байтове. Методите на този клас и най-вече **System.in.read()** се използват за четене от стандартния вход.

В чистия му вариант **System.in.read()** почти не се използва, защото има абстракции, които улесняват четенето от конзолата. В последните версии на Java се появяват допълнителни улеснения за четене, но идеологията винаги си остава една и съща.



Тази схема показва взаимодействието между трите стандартни променливи предназначени за вход (**STDIN**) и изход (**STDOUT, STDERR**) в **Java**. **STDERR** е стандартният изход за грешки. На него може да печатаме по следния начин:

```
System.err.println("This is error!");
```

Ще разгледаме още няколко примера за вход и изход от конзолата.

Печатане на писмо

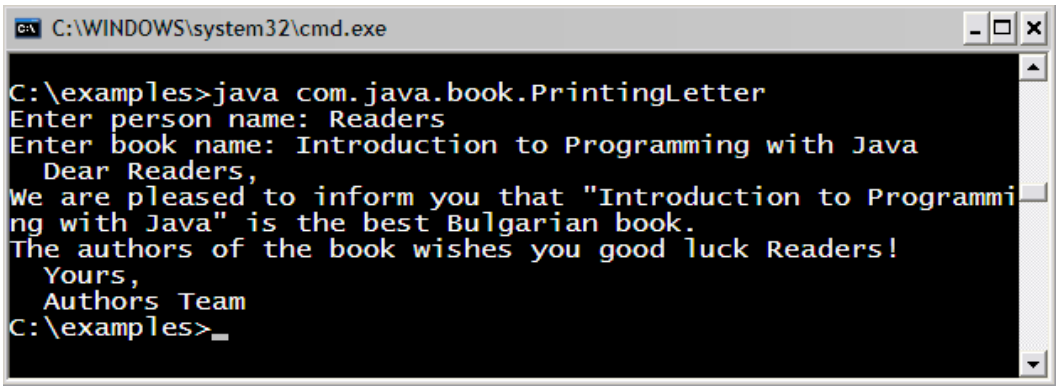
Това е един практичен пример показващ конзолен вход и форматиран текст под формата на писмо.

PrintingLetter.java

```
publicclass PrintingLetter {

    publicstaticvoid main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.printf("Enter person name: ");
        String person = input.nextLine();

        System.out.printf("Enter book name: ");
        String book = input.nextLine();

        String from = "Authors Team";

        System.out.printf("  Dear %s,%n", person);
        System.out.printf("We are pleased to inform " +
```

```
                    "you that \"%2$s\" is the best Bulgarian book. \n" +
                    "The authors of the book wishes you good luck %s!%n",
                    person, book);

            System.out.println("  Yours,");
            System.out.printf("  %s", from);
        }
}
```

Резултатът от изпълнението на горната програма би могъл да следния:



В този пример имаме предварителен шаблон на писмо. Програмата "задава" няколко въпроса на потребителя и по този начин прочита от конзолата нужната информация, за да отпечата писмото, като замества форматиращите спецификатори с попълнените от потребителя параметри. Следва печатане на цялото писмо в конзолата.

Упражнения

1. Напишете програма, която чете от конзолата три числа от тип **int** и отпечатва тяхната сума.
2. Напишете програма, която чете от конзолата радиуса "**r**" на кръг и отпечатва неговия периметър и обиколка.

3. Дадена фирма има име, адрес, телефонен номер, факс номер, уеб сайт и мениджър. Мениджърът има име, фамилия и телефонен номер. Напишете програма, която чете информацията за компанията и нейния мениджър и я отпечатва след това на конзолата.
4. Напишете програма, която чете от конзолата две цели числа (**integer**) и отпечатва, колко числа има между тях, такива, че остатъкът им от деленето на 5 да е 0.

7. Условни конструкции.

Условни конструкции if и if-else

Условните конструкции **if** и **if**-else представляват тип условен контрол, чрез който вашата програма може да се държи различно, в зависимост от приложения условен тест.

Условна конструкция if

Основният формат на условната конструкция **if** е, както следва:

```
if (булев израз) {
```

```
        тяло на условната конструкция
}
```

Форматът включва: **if**-клауза, булев израз и тяло на условната конструкция.

Булевият израз може да бъде променлива от булев тип, булев логически израз или релационен израз. Булевият израз не може да бъде цяло число.

Тялото е онази част, заключена между големите къдрави скоби: **{}**. То може да се състои от един или няколко оператора. Когато се състои от няколко оператора, говорим за съставен блоков оператор. Също така в тялото могат да бъдат включвани една или няколко конструкции.

Изразът в скобите трябва да бъде сведен до булева стойност **true** или **false**. Ако изразът бъде изчислен до стойност **true**, тогава се изпълнява тялото на условната конструкция. Ако пък от друга страна, резултатът от изчислението на булевия израз е **false**, то операторите в тялото няма да бъдат изпълнени.

Условна конструкция if – пример

Пример за условна конструкция **if**:

```
publicstaticvoid main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter two numbers.");
        int firstNumber = input.nextInt();
        int secondNumber = input.nextInt();
        int biggerNumber = firstNumber;
        if (secondNumber > firstNumber) {
                biggerNumber = secondNumber;
        }
        System.out.printf("The bigger number is: %d%n", biggerNumber);
}
```

Конструкцията if и къдравите скоби

При наличието на само един оператор в тялото на **if**-конструкцията, къдравите скоби, обозначаващи тялото на условния оператор могат да бъдат изпуснати, както е показано по-долу. Добра практика е, обаче те да бъдат поставяни, дори при наличието на само един оператор. Целта е програмния код да бъде по-четим.

```
int a = 6;
if (a > 5)
        System.out.println("Променливата a е по-голяма от 5.");
        System.out.println("Този код винаги ще се изпълни!");
```

В горния пример кодът е форматиран заблуждаващо и създава впечатление, че и двете печатания по конзолата се отнасят за тялото на **if**блока, а всъщност това е вярно само за първия от тях.

Винаги слагайте къдрави скоби {} за тялото на if блоковете дори ако то се състои само от един оператор!

Условна конструкция if-else

Основният формат на условната конструкция **if-else** е, както следва:

```
if (булев израз) {
        тяло на условната конструкция
} else {
        тяло на else-конструкция
}
```

Форматът включва: запазена дума **if**, булев израз, тяло на условната конструкция, запазена дума **else**, тяло на **else**-конструкция. Тялото на **else**-конструкцията може да се състои от един или няколко оператора.

Изчислява се изразът в скобите (булевият израз). Резултатът от изчислението може да бъде сведен до **true** или **false**. В зависимост от резултата са възможни

два пътя, по които да продължи потока от изчисления. Ако булевият израз се сведе до true, то се изпълнява тялото на условната конструкция, а тялото на **else**-конструкцията се пропуска и операторите в него не се изпълняват. От друга страна, ако булевият израз се сведе до **false**, то се изпълнява тялото на **else**-конструкцията, а тялото на условната конструкция се пропуска и операторите в него не се изпълняват.

Условна конструкция if-else – пример

Нека разгледаме следния програмен код:

```
x = 3;
if (x > 3) {
        System.out.println("x е  по-голямо от 3");
} else {
        System.out.println("x не е по-голямо от 3");
}
```

Програмният код може да бъде интерпретиран по следния начин: Ако x>3, то резултатът на изхода е: "x е по-голямо от 3", иначе (**else**) резултатът е: "x не е по-голямо от 3". В случая, понеже x=3, след изчислението на булевия израз ще бъде изпълнен операторът от **else**-конструкцията. Резултатът от примера е:

```
x не е по-голямо от 3
```

Вложени if конструкции

Понякога е нужно програмната логика в дадена програма или приложение да бъде представена посредством if-конструкции, които се съдържат една в друга. Наричаме ги **вложениif** или **if**-else конструкции.

Влагане наричаме поставянето на **if** или **if-else** клауза в друга **if** или **else** конструкция. Всяка **else** клауза се отнася за най-близко разположената предходна **if** клауза. По този начин разбираме коя **else** клауза към коя **if** клауза се отнася.

Не е добра практика нивото на влагане да бъде повече от три, тоест не трябва да бъдат влагани повече от три условни конструкции една в друга.

Ако поради една или друга причина се наложи да бъде направено влагане на повече от три конструкции, то трябва да се търси проблем в архитектурата на създаваната програма или приложение.

Вложени if конструкции – пример

Пример за употреба на вложени **if** конструкции:

```
Scanner input = new Scanner(System.in);
System.out.println(
"Please enter two numbers (on separate lines).");
int first = input.nextInt();
int second = input.nextInt();
if (first == second) {
        System.out.println("These two numbers are equal.");
} else {
        if (first > second) {
                System.out.println("The first number is greater.");
        } else {
                System.out.println("The second number is greater.");
        }
}
```

В примера се въвеждат две числа и се сравняват на две стъпки: първо се сравняват дали са равни и ако не са, се сравняват отново, за да се установи кое от числата е по-голямо. Ето примерен резултат от работата на горния код:

```
Please enter two numbers (on separate lines).
2
```

```
4
The second number is greater.
```

Условна конструкция switch-case

В следващата секция ще бъде разгледана условната конструкция **switch** за избор измежду списък от възможности.

Как работи switch-case конструкцията?

Конструкцията switch прави избор измежду части от програмен код на базата на изчислената стойност на определен целочислен израз (целочислен селектор). Форматът на конструкцията за избор на вариант е:

```
switch (целочислен селектор) {
        case целочислена-стойност-1: конструкция; break;
        case целочислена-стойност-2: конструкция; break;
        case целочислена-стойност-3: конструкция; break;
        case целочислена-стойност-4: конструкция; break;
        default: конструкция;}
```

Целочисленият селектор е израз, даващ като резултат целочислена стойност. Операторът **switch** сравнява резултата от целочисления селектор с всяка една целочислена стойност (етикет). Ако се открие съвпадение, се изпълнява съответната конструкция (проста или съставна). Ако не се открие съвпадение, се изпълнява **default** конструкцията. Стойността на целочисления израз трябва задължително да бъде изчислена преди да се сравнява с целочислените стойности вътре в **switch** конструкцията.

Виждаме, че в горната дефиниция всеки case завършва с **break**, което води до преход към края на тялото на **switch**. Това е стандартният начин за изграждане на **switch** конструкция, но **break** е незадължителна клауза. Ако липсва, кодът

след **case** конструкцията, при която е срещнато съвпадение между стойността на селектора и на етикета, ще се изпълни и след това ще бъдат изпълнени конструкциите на другите case-оператори надолу, независимо че стойностите на техните етикети не биха съвпаднали със стойността на селектора. Това продължава докато бъде достигната break клауза след някоя case конструкция. Ако такава не бъде достигната, изпълнението ще продължи до достигане на края на **switch**.

От дадената дефиниция на конструкцията **switch** забелязваме, че след конструкцията на **default** липсва **break**. Това няма значение, защото в случая след **default** конструкцията няма други **case** конструкции за изпълнение, а се бележи края на тялото на **switch**. Програмистът може да постави след **default** конструкцията **break**, ако това е важно за добрия стил или за потока на изчисленията. Трябва да се има предвид, че не е задължително **default** конструкцията да е на последно място, тя може да бъде поставена някъде във вътрешността на switch конструкцията.

Правила за израза в switch

Конструкцията **switch** е един ясен начин за имплементиране на избор между множество варианти (тоест, избор между няколко различни пътища за изпълнение). Тя изисква селектор, който се изчислява до цяло число от типа **int, byte, char** или **enum**. Ако искаме да използваме, например, низ или число с плаваща запетая като селектор, това няма да работи в **switch** конструкция. За нецелочислени типове данни трябва да използваме последователност от **if** конструкции.

Използване на множество етикети

Използването на множество етикети е удачно, когато искаме да бъде изпълнена една и съща конструкция в повече от един случай. Нека разгледаме следния пример:

```
int number = 6;
```

```
switch (number) {
        case 1:
        case 4:
        case 6:
        case 8:
        case 10: System.out.println("Числото не е просто!"); break;
        case 2:
        case 3:
        case 5:
        case 7: System.out.println("Числото е просто!"); break;
        default: System.out.println("Не знам какво е това число!");
}
```

В този пример е имплементирано използването на множество етикети чрез написването на case конструкции без **break** след тях, така че в случая първо ще се изчисли целочислената стойност на селектора – тук тя е 6, и след това тази стойност ще започне да се сравнява с всяка една целочислена стойност в **case** конструкциите. Въпреки, че ще бъде срещнато съвпадение със стойността на селектора и третата case конструкция, изпълнението ще продължи надолу до срещането на първия **break**.

Резултатът от предходния програмен код е:

```
Числото не е просто
```

Упражнения

1. Да се напише **if**-конструкция, която изчислява стойността на две целочислени променливи и разменя техните стойности, ако стойността на първата променлива е по-голяма от втората.

2. Напишете програма, която показва знака (+ или -) от частното на две реални числа, без да го пресмята.
3. Напишете програма, която намира най-голямото по стойност число, измежду три дадени числа.
4. Напишете програма, която намира най-голямото по стойност число измежду дадени 5 числа.
5. Дадени са няколко цели числа. Напишете програма, която намира онези подмножества от тях, които имат сума 0. Примери:
   - Ако са дадени числата {-2, -1, 1}, сумата на -1 и 1 е 0.
   - Ако са дадени числата {3, 1, -7}, няма подмножества със сума 0.
6. Напишете програма, която прилага бонус точки към дадени точки в интервала [1..9] чрез прилагане на следните правила:
   - Ако точките са между 1 и 3, програмата ги умножава по 10.
   - Ако точките са между 4 и 6, ги умножава по 100.
   - Ако точките са между 7 и 9, ги умножава по 1000.
   - Ако точките са 0 или повече от 9, се отпечатва съобщение за грешка.

8. Цикли.

Какво е "цикъл"?

В програмирането често се налага многократното изпълнение на дадена последователност от операции. **Цикълът (loop)** е структурата, която позволява това изпълнение без излишно писане на повтарящ се код. В зависимост от вида на цикъла, програмния код в него се повтаря:

- определени от фиксирано число пъти;
- докато дадено условие е изпълнено.

Цикъл, който никога не свършва, се нарича **безкраен(infinite)**.

Конструкция за цикъл while

Един от най-простите и често използвани цикли е **while**.

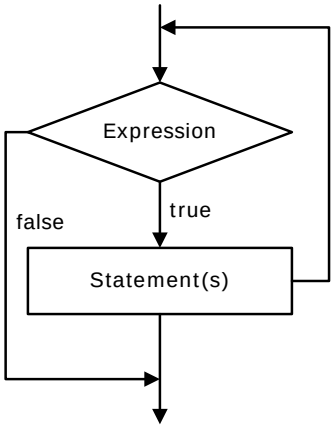
```
while (condition) {
  statements;
}
```

Condition е израз, който връща булев резултат – **true** или **fasle**. Той определя докога ще се изпълнява тялото на цикъла и се нарича – **loop condition.**

Statements са програмният код, изпълняван в цикъла. Те се наричат тяло на цикъла.

При **while** цикъла първо се изпълнява булевия израз, ако резултатът от него е true, се изпълнява и последователността от операции, това се повтаря докато (**pre-test loop**). Ето логическата схема, по която се изпълняват **while**

циклите:



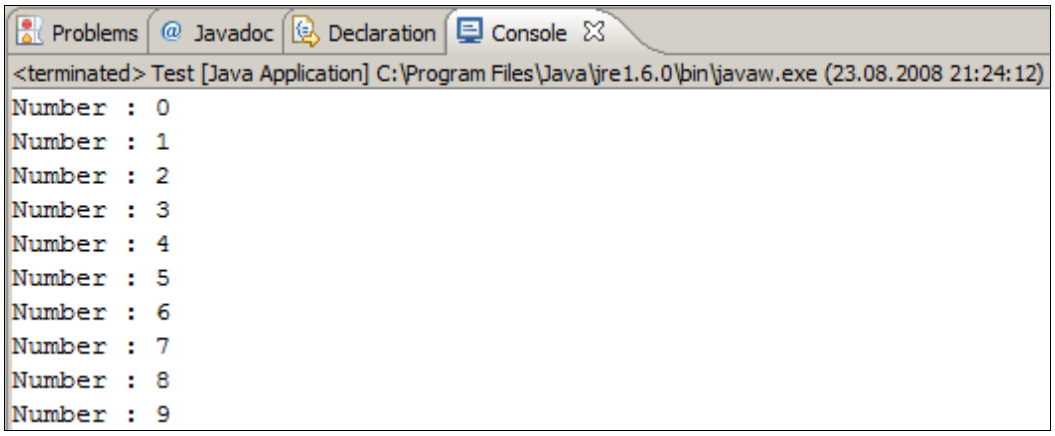
Използване на while цикли

While циклите изпълняват група операции докато е в сила дадено условие.

Нека разгледаме един съвсем прост пример за използването на **while** цикъл, при който само се отпечатват на конзолата числата в интервала от 0 до 9 в нарастващ ред:

```
int counter = 0;
while (counter < 10) {
        System.out.printf("Number : %d%n", counter);
        counter++;
}
```

При изпълнение на примерния код получаваме следния резултат:



Нека дадем още примери, за да се убедите в ползата от циклите и да ви покажем какви задачи можем да решаваме с цикли.

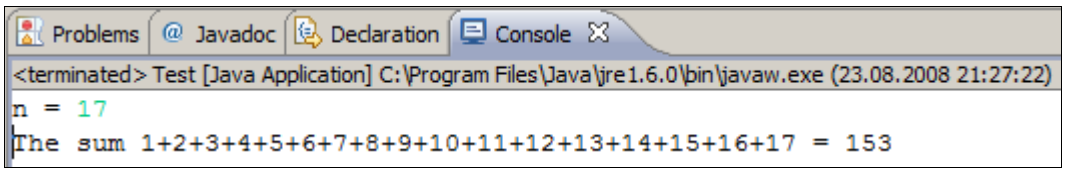
Сумиране на числата от 1 до N – пример

В настоящия пример ще разгледаме как с помощта на цикъла **while** се намира сумата на числата от **1** до **N**. Числото **N** се чете от конзолата. Инициализираме променливите **num** и **sum** със стойност **1**. В **num** ще пазим текущото число, което ще добавяме към сумата на предходните. При всяко преминаване през цикъла ще го увеличаваме с **1**, за да получим следващото число, след което при влизане в цикъла проверяваме дали то отговаря на условието на цикъла, тоест

дали е в интервала от **1** до **N**. **Sum** е променливата за сумата на числата. При влизане в цикъла добавяме към нея поредното число записано в **num**. На конзолата принтираме всички **num** (числа от 1 до N) с разделител "+" и крайния резултат от сумирането след приключване на цикъла.

```
Scanner input = new Scanner(System.in);
System.out.print("n = ");
int n = input.nextInt();
int num = 1;
int sum = 1;
System.out.print("The sum 1");
while (num < n) {
        num++;
        sum += num;
        System.out.printf("+%d", num);
}
System.out.printf(" = %d%n", sum);
```

Изходът е програмата е следният:



Оператор break

Операторът **break** се използва за излизане от цикъла. Операциите в цикъла се изпълняват в съответния им ред и при достигане на оператора **break**, независимо дали условието за излизане от цикъла е изпълнено, изпълнението на цикъла се прекратява, като кода след **break** не се изпълнява.

Изчисляване на факториел – пример

В този пример ще пресметнем факториела на въведено през конзолата число с помощта на безкраен **while** цикъл и оператора **break**. Да си припомним от математиката какво е факториел и как се изчислява. Това е функция на естествено число n, която изразява произведението на всички естествени числа, по-малки или равни на n. Записва се n! и по дефиниция:

- n! = 1*2*3.....(n-1)*n, за n>1;
- 1! = 1;
- 0! = 1.

N! може да се изрази чрез факториел от естествени числа, по-малки от n:

- n! = (n-1)!n, като използваме началната стойност 1! = 1.

Това ще използваме и ние, за да изчислим факториела на n. Инициализираме променливата **factorial** с 1, а n – четем от конзолата. **While** цикълът, който ще конструираме, искаме да е безкраен. За тази цел условието трябва винаги да е **true**. Ще използваме оператора **break**, за да прекратим цикълът, когато n достигне 1. Самото изчисление ще започнем от числото **n**, с него ще умножаваме **factorial**, след което **n** ще намаляваме с 1. Или ще сметнем факториел по следната формула: n*(n-1)*(n-2)*…*3*2, при n=1 прекратяваме изпълнението на цикъла.

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
long factorial = 1;
while (true) {
        if (n == 1) {
                break;
        }
        factorial *= n;
```

```
        n--;
}
System.out.println("n! = " + factorial);
```

Ако въведем 10 като вход, на конзолата ще видим следния резултат:

```
10
n! = 3628800
```

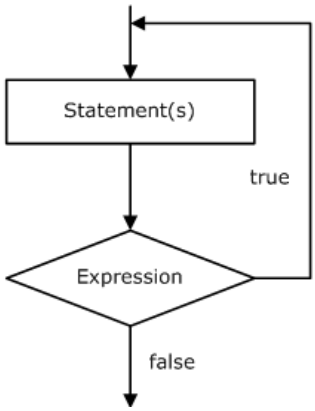
Конструкция за цикъл do-while

Do-while цикълът е аналогичен на **while** цикъла, само че при него проверката на булевия израз се прави след изпълнението на операциите в цикъла. Този тип цикли се наричат – цикли с условие в края (post-test loop).

Ето как изглежда един **do-while**цикъл:

```
do {
  statements;
}
while (expression);
```

Схематично **do-while**циклите се изпълняват по следната логическа схема:



Използване на do-while цикли

Do-while цикълът се използва, когато искаме да си гарантираме, че поредицата от операции в него ще бъде многократно, но май-малко веднъж.

Изчисляване на факториел – пример

В този пример отново ще изчислим факториела на дадено число **n**, но този път вместо безкраен **while** цикъл, ще използваме **do-while**. Логиката е аналогична на тази в предния пример. Умножаваме всяко следващо число с произведението на предходните и го намаляваме с 1, след което проверяваме дали то все още е по-голямо от 0. Накрая отпечатваме получения резултат на конзолата.

```
Scanner input = new Scanner(System.in);
System.out.print("n = ");
int n = input.nextInt();
long factorial = 1;
do {
        factorial *= n;
        n--;
} while (n > 0);
System.out.println("n! = " + factorial);
```

Ето го и резултатът от изпълнение на горния пример при n=7:

```
n = 7
n! = 5040
```

Конструкция за цикъл for

For-**циклите** са малко по-сложни от **while**и **do-while**циклите, но за сметка на това могат да решават по-сложно задачи с по-малко писане на код. Характерната за **for**-цикъла структура е следната:

```
for (initialization; test; update) {
        statements;
}
```

Тя се състои от инициализационна част за брояча, булево условие, израз за обновяване на брояча и тяло на цикъла. Броячът на **for** цикъла го отличава от другите цикли. Броят на итерациите на този цикъл най-често се знае още преди да започне изпълнението му.

Безкраен цикъл (infinite loop) чрез оператора **for** се конструира по следния начин:

```
for ( ; ; ) {
        statements;
}
```

Безкраен цикъл означава цикъл, който никога не завършва. Обикновено в нашите програми нямаме нужда от безкраен цикъл, освен, ако някъде в тялото му не използваме **break**, за да завършим цикъла преждевременно.

Инициализация на for цикъла

For-циклите могат да имат инициализационен блок:

```
for (int num = 0; ...; ...) {        ……      }
```

Той се изпълнява веднъж, точно преди влизане в цикъла. Обикновено се използва за деклариране на променливата-брояч. Тази променлива е "видима" и може да се използва само в рамките на цикъла.

Условие на for цикъла

For-циклите могат да имат условие за повторение:

```
for (int num = 0; num < 10; ...) {
}
```

То се изпълнява веднъж, преди всяка итерация на цикъла. При резултат **true** се изпълнява тялото на цикъла, а при **false** то се пропуска и се преминава към останалата част от програмата. Използва се като **loop condition** (условие на цикъла).

Обновяване на водещата променлива

Последната част от един for-цикъл съдържа код, който обновява водещата променлива:

```
for (int num = 0; num < 10; num++) {
}
```

Той се изпълнява след като е приключило изпълнението на тялото на цикъла. Най-често се използва за обновяване стойността на брояча.

For-цикъл с няколко променливи

С конструкцията за **for**-цикъл можем да ползваме едновременно няколко променливи. Ето един пример, в който имаме два брояча. Единият се движи от 1 нагоре, а другият се движи от 10 надолу:

```
for (int small=1, large=10; small<large; small++, large--) {
        System.out.printf("%d %d\n", small, large);
}
```

Условието за прекратяване на цикъла е застъпване на броячите. В крайна сметка се получава следния резултат:

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

Оператор continue

Операторът **continue** спира текущата итерация на най-вътрешния цикъл, но не излиза от него. С помощта на следващия пример ще разгледаме как точно се използва този оператор.

Ще намерим сумата на всички нечетни естествени числа в интервала [1...n], който не се делят на 7. Ще използваме **for**-цикъл. При инициализиране на променливата ще и зададем стойност 1, тъй като това е първото нечетно естествено число в интервала [1...n]. Ще проверяваме дали **i** е все още в интервала (**i <= n**). В израза за обновяване на променливата ще я увеличаваме с 2, за да работим само с нечетни числа. В тялото на цикъла ще правим проверка дали числото се дели на 7. Ако това е изпълнено извикваме оператора **continue**, който ще прекрати изпълнението на цикъла (няма да добави текущото число към сумата). Ще се извърши обновяване на променливата и ще продължи изпълнението на цикъла. Ако не е изпълнено ще премине към обновяване на сумата с числото.

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
int sum = 0;
for (int i = 1; i <= n; i += 2) {
        if (i % 7 == 0) {
                continue;
        }
        sum += i;
}
System.out.println("sum = " + sum);
```

Резултатът от примера при n=26 еледният:

```
26
```

```
sum = 141
```

Разширена конструкция за цикъл for

От версия 5 на Javaза удобство на програмистите беше добавена още една конструкция за цикъл, т.нар. **foreach**конструкция, наричана още разширен **for**-цикъл. Тази конструкция служи за обхождане на всички елементи на даден масив, списък или колекция от елементи. Подробно с масивите ще се запознаем в темата "Масиви", но за момента можем да си представяме един масив като последователност от числа или други елементи.

Ето как изглежда един разширен **for**-цикъл:

```
for (variable : collection) {
        statements;
}
```

Както виждате, той е значително по-прост от стандартния **for**-цикъл и затова много-често се предпочита от програмистите, тъй като спестява писане.

Ето един пример, в който ще видите разширения **for**-цикъл в действие:

```
int[] numbers = {2, 3, 5, 7, 11, 13, 17, 19};
for (int i : numbers) {
        System.out.printf("%d ", i);
}
System.out.println();

String[] towns = {"Sofia", "Plovdiv", "Varna", "Bourgas"};
for (String town: towns) {
```

```
        System.out.printf("%s ", town);
}
```

В примера се създава масив от числа и след това те се обхождат с разширения for-цикъл и се отпечатват на конзолата. След това се създава масив от имена на градове (символни низове) и по същия начин се отпечатват на конзолата. Ето какъв е резултатът от примера:

```
2 3 5 7 11 13 17 19
Sofia Plovdiv Varna Bourgas
```

Вложени цикли

Вложените цикли представляват конструкция от няколко цикъла един в друг. Най-вътрешния цикъл се изпълнява най-много пъти. В примерната конструкция по долу ще разгледаме пример за вложен цикъл. След инициализация на първия **for** цикъл ще започне да се изпълнява втория. Ще се инициализира променливата му, ще се провери условието и ще се изпълнят изразите в него, след което ще се обнови променливата и ще продължи изпълнението на този цикъл, докато условието му не върне **false**. В този случай ще се върне в първия **for** цикъл, ще се извърши обновяване на неговата променлива и отново ще бъде изпълнен целия втори цикъл. Обикновено 2 **for** цикъла се използват за манипулация на двумерни масиви. Вложените цикли, използвани необмислено, могат да влошат производителността на програмата.

```
for (initialization; test; update) {
        for (initialization; test; update) {
                statements;
        }
        …
}
```

Отпечатване на триъгълник – пример

Нека си поставим следната задача: по дадено число **n**да отпечатаме на конзолата триъгълник с **n** на брой реда, изглеждащ по следния начин:

```
1
1 2
1 2 3
. . .
1 2 3 . . . n
```

Това ще направим с два **for**-цикъла. Външния ще ни обхожда редовете, а вътрешния – елементите в тях. Когато сме на първия ред, трябва да отпечатаме "1" (1 елемент, 1 итерация на вътрешния цикъл). На втория – "1 2" (2 елемента, 2 итерации). Виждаме, че има зависимост между реда, на който сме и броя на елементите, който ще отпечатваме. Това ни дава информация за определяне конструкцията на вътрешния цикъл:

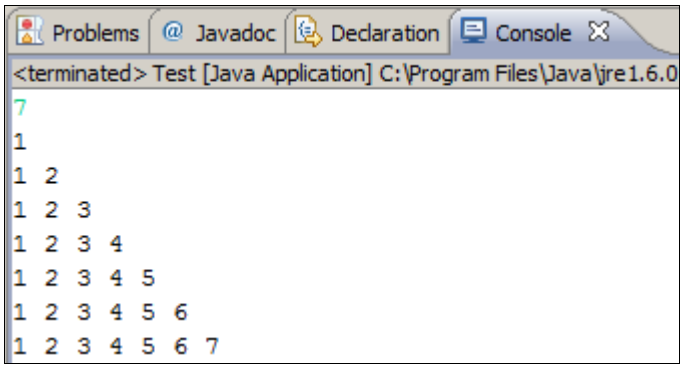
- инициализираме променливата с 1 (първото число, което ще отпечатаме) =>**col** = 1;
- условието ни зависи от реда, на който сме. Той ограничава елементите =>**col**<= **row**;
- обновяваме променливата увеличавайки я с 1.

На практика трябва да направим един **for**-цикъл (външен) от 1 до **n** (за редовете) и в него още един **for**-цикъл (вътрешен) за числата в текущия ред, който да е от 1 до номера на текущия ред. Външният цикъл ходи по редовете, а вътрешният – по всяка от колоните за текущия ред. В крайна сметка получаваме следния сорс код:

```
Scanner input = new Scanner(System.in);
int n = input.nextInt();
for (int row = 1; row <= n; row++) {
```

```
        for (int col = 1; col <= row; col++) {
                System.out.print(col + " ");
        }
        System.out.println();
}
```

Ако го изпълним, ще се убедим, че работи коректно. Ето как изглежда резултатът при n=7:



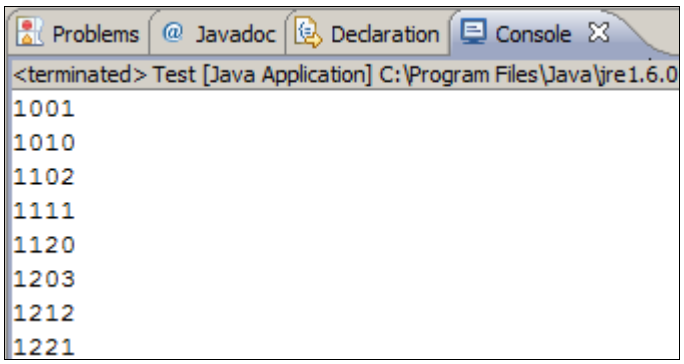
Щастливи числа – пример

Нека разгледаме още един пример, с който ще покажем, че можем да влагаме и повече от два цикъла един в друг. Целта е да намерим и отпечатаме всички четирицифрени числа от вида ABCD, за които: A+B = C+D (наричаме ги щастливи числа). Това ще реализираме с помощта на четири **for**-цикъла – за всяка цифра по един. Най-външния цикъл ще ни определя хилядните. Той ще започва от 1, а останалите от 0. Ще правим проверка, дали текущото ни число е щастливо, в най-вътрешния цикъл. Ако е така ще го отпечатваме на конзолата.

```
for (int a = 1; a <= 9; a++) {
        for (int b = 0; b <= 9; b++) {
                for (int c = 0; c <= 9; c++) {
                        for (int d = 0; d <= 9; d++) {
```

```
                        if ((a + b) == (c + d)) {
                            System.out.printf("%d%d%d%d%n", a,
b, c, d);
                        }
                    }
                }
            }
}
```

Ето част от отпечатания резултат (целият е много дълъг):



Упражнения

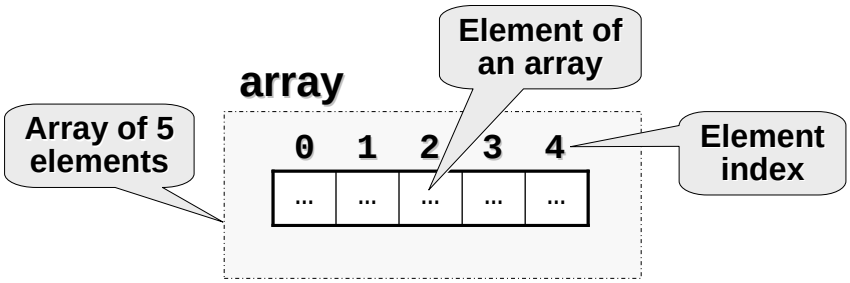
1. Напишете програма, която отпечатва на конзолата числата от 1 до N. Числото N се чете от стандартния вход.
2. Напишете програма, която отпечатва на конзолата числата от 1 до N, които не се делят на 3 и 7. Числото N се чете от стандартния вход.
3. Напишете програма, която чете от конзолата поредица от цели числа и отпечатва най-малкото и най-голямото от тях.
4. Напишете програма, която отпечатва всички възможни карти от стандартно тесте без джокери (имаме 52 карти: 4 бои по 13 карти).
5. Напишете програма, която чете от конзолата числото N и отпечатва сумата на първите N члена от редицата на Фибоначи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

6. Напишете програма, която пресмята N!/K! за дадени N и K (1<K<N).

9. Масиви.

Какво е "масив"?

Масивите са неизменна част от езиците за програмиране. Те представляват съвкупности от променливи, които наричаме **елементи**:



Елементите на масивите са номерирани с числата 0, 1, 2, ... N-1. Тези номера на елементи се наричат **индекси**. Броят елементи в даден масив се нарича **дължина на масива.**

Всички елементи на даден масив са от един и същи тип, независимо дали е **примитивен** или **референтен**. Това ни помага да представим група от еднородни елементи като подредена свързана последователност и да ги обработваме като едно цяло.

Масивите могат да бъдат от различни размерности, като най-често използвани са **едномерните** и **двумерните** масиви. Едномерните масиви се наричат още вектори, а двумерните матрици.

Деклариране и заделяне на масиви

В Java масивите имат фиксирана дължина, която се указва при инициализирането му и определя броя на елементите му. След като веднъж сме задали дължината на масив не е възможно да я променяме.

Деклариране на масив

Масиви в Java декларираме по следния начин:

```
int[] myArray;
```

Тук променливата **myArray** е името на масива, ко йто е от тип (**int[]**) т.е. декларирали сме масив от цели числа. С **[]** се обозначава, че променливата, която декларираме ще е масив, а не единичен елемент.

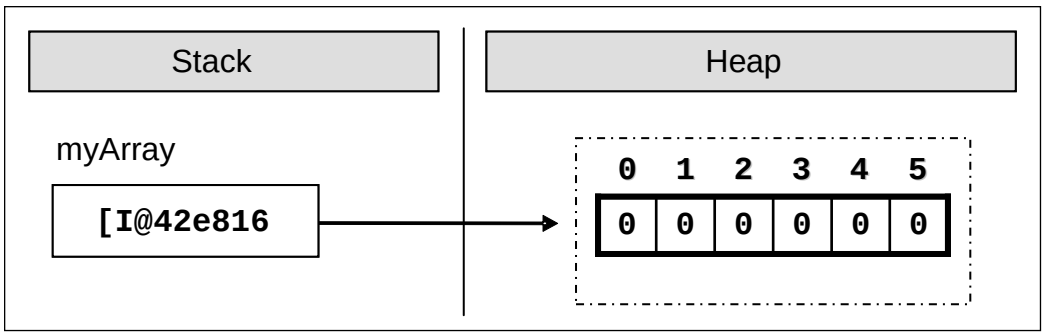
При декларация името на променливата, която е от тип масив, представлява референция (**reference**), която сочи към **null**, тъй като още не е заделена памет за елементите на масива.

Създаване (заделяне) на масив – оператор new

В Java масив се създава с ключовата дума **new**, която служи за заделяне (алокиране) на памет:

```
int[] myArray = newint[6];
```

В примера заделяме масив с размер 6 елемента от целочислен тип. Това означава, че в динамичната памет (heap) се заделя участък от 6 последователни цели числа:



Картинката показва, че след заделянето на масива променливата **myArray** сочи някакъв адрес (**0x42e816**) в динамичната памет, където се намира нейната стойност. Елементите на масивите винаги се съхраняват в динамичната памет (в т. нар.**heap**).

Инициализация на масив. Стойности по подразбиране

Преди да използваме елемент от даден масив той трябва да има начална стойност. В някои езици за програмиране не се задават начални стойности по подразбиране, и тогава при опит за достъпване да даден елемент възниква грешка. В Java всички променливи, включително и елементите на масивите имат начална стойност по подразбиране (default initial value)> Тази стройност е равна на 0 при числените типове или неин еквивалент при нечислени типове (например **null**за обекти и **false** за булевия тип).

Разбира се, начални стойности можем да задаване и изрично. Това може да стане по различни начини. Един възможен е чрез използване на литерален израз за елементите на масива (**array literal expression**):

```
int[] myArray = {1, 2, 3, 4, 5, 6};
```

В този случай създаваме и инициализираме масива едновременно.

Достъп до елементите на масив

Достъпът до елементите на масивите е пряк, по индекс. Всеки елемент може да се достъпи с името на масива и съответния му индекс, поставен в квадратни скоби. Можем да осъществим достъп до даден елемент както за четене така и за писане т.е. да го третираме като обикновена променлива.

Масивите могат да се обхождат с помощта на някоя от структурите за **цикъл**, като най-често използван е класическият **for** цикъл.

Пример за достъп до елемент на масив:

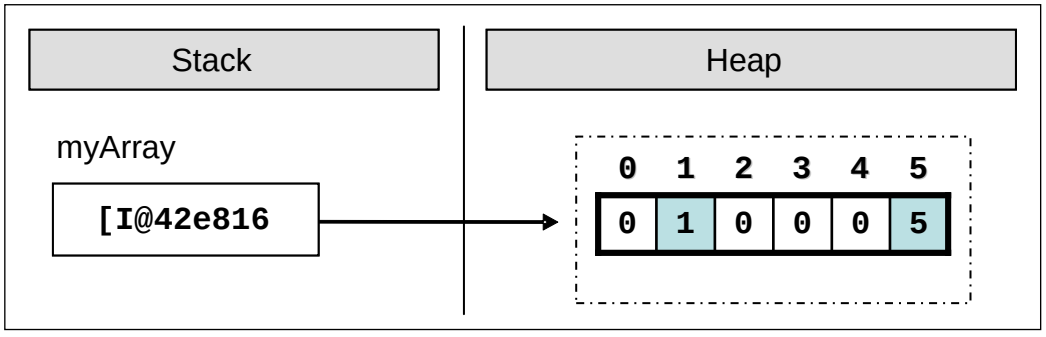
```
myArray[index] = 100;
```

В горния пример присвояваме стойност 100 на елемента, намиращ се на позиция **index**, където **index** е валиден за масива индекс.

Ето един пример, в който заделяме масив от числа и след това променяме някои от елементите му:

```
int[] myArray = newint[6];
myArray[1] = 1;
myArray[5] = 5;
```

След промяната на елементите, масивът се представя в паметта по следния начин:



Граници на масив

Масивите обикновено са **нулево-базирани**, т.е. номерацията на елементите започва от **0**. Първият елемент има индекс 0, вторият 1 и т.н. Ако един масив има **N** елемента, то последният елемент се намира на индекс **N-1.**

Излизане от границите на масив

Достъпът до елементите на масивите се проверява по време на изпълнение от виртуалната машина на Javaи тя не допуска излизане извън границите и размерностите им. При всеки достъп до елемент на масива по се прави проверка, дали индексът е валиден или не. Ако не е се хвърля изключение от тип**java.lang.ArrayIndexOutOfBoundsException**. Естествено, тази проверка си има и своята цена и тя е леко намаляване на производителността.

Четене на масив от конзолата

Нека разгледаме как можем да прочетем стойностите на масив от конзолата. Ще използваме **for** цикъл и средствата на Java за четене на числа от конзолата.

Първоначално, за да заделим памет за масива, може да прочетем цяло число **n**от конзолата и да го ползваме като негов размер:

```
int n = input.nextInt();
int[] array = newint[n];
```

Отново използваме цикъл, за да обходим масива. На всяка итерация присвояваме на текущия елемент прочетеното от конзолата число. Цикъла ще се завърти **n** пъти т.е. ще обходи целия масив и така ще прочетем стойност за всеки елемент от масива:

```
for (inti = 0; i < n; i++) {
  array[i] = input.nextInt();
}
```

Отпечатване на масив на конзолата

Често се налага след като сме обработвали даден масив да изведем елементите му на конзолата, било то за тестови или други цели.

Отпечатването на елементите на масив става по подобен начин на инициализирането на елементите му, а именно като използваме цикъл, който обхожда масива. Няма строги правила за извеждането на данните. Разбира се, добра практика е те да бъдат добре форматирани.

Често срещана грешка е опит да се изведе на конзолата масив директно, по следния начин:

```
String[] array = { "one", "two", "three", "four" };
System.out.println(array);
```

Този код за съжаление не отпечатва съдържанието на масива, а неговия адрес в динамичната памет (защото масивите са референтни типове). Ето как изглежда резултатът от изпълнението на горния код:

```
[Ljava.lang.String;@42e816
```

За да изведем коректно елементите на масив на конзолата можем да използваме **for** цикъл:

```
String[] array = { "one", "two", "three", "four" };
for (int index = 0; index < array.length; index++) {
        System.out.printf("element[%d] = %s%n", index, array[index]);
}
```

Обхождаме масива с цикъл **for**, който извършва **array.length** на брой итерации, с помощта на метода **System.out.printf**, извеждаме данните на конзолата във **форматиран** вид. Резултатът е следният:

```
element[0] = one
element[1] = two
```

```
element[2] = three
element[3] = four
```

Има и още един, по-лесен начин да отпечатаме съдържанието на масив:

```
String[] array = { "one", "two", "three", "four" };
System.out.println(java.util.Arrays.toString(array));
```

Резултатът е добре форматиран символен низ, съдържащ всички елементи на масива, изброени със запетайка:

```
[one, two, three, four]
```

Итерация с for цикъл

Добра практика е да използваме **for** цикъл при работа с масиви и изобщо при индексирани структури. Ето един пример, в който удвояваме стойността на всички елементи от даден масив с числа:

```
int[] array = newint[] {1, 2, 3, 4, 5};
for (int index = 0; index < array.length; index++) {
        array[index] = 2 * array[index];
}

System.out.println(Arrays.toString(array));
```

Чрез **for** цикъла можем да имаме постоянен поглед върху текущия индекс на масива и да достъпваме точно тези елементи, от които имаме нужда. Итерирането може да не се извършва последователно т.е. индексът, който **for** цикъла ползва може да прескача по елементите според нуждите на нашия алгоритъм. Например можем да обходим част от даден масив, а не целия. Ето един пример:

```
for (int index = 0; index < array.length; index += 2) {
  array[index] = array[index] * array[index];
}
```

В горния пример обхождаме всички елементи на масива, намиращи се на четни позиции и повдигаме на квадрат стойността във всеки от тях.

Понякога е полезно да обходим масив отзад напред. Можем да постигнем това по напълно аналогичен начин, с разликата, че **for** цикълът ще започва с начален индекс, равен на индекса на последния елемент на масива, и ще се намаля на всяка итерация. Ето един такъв пример:

```
int[] array = newint[] {1, 2, 3, 4, 5};
System.out.print("Reversed: ");
for (int i = array.length - 1; i >= 0; i--) {
  System.out.print(array[i] + " ");
}
```

В горния пример обхождаме масива от зад напред последователно и извеждаме всеки негов елемент на конзолата.

Многомерни масиви

До момента разгледахме работата с едномерни масиви, известни в математиката като "вектори". В практиката, обаче, често се ползват масиви с повече от едно измерения. Например стандартна шахматна дъска се представя лесно с двумерен масив с размер 8 на 8 (8 полета в хоризонтална посока и 8 полета във вертикална посока).

Какво е "многомерен масив"? Какво е "матрица"?

Всеки допустим в Java тип може да бъде използван за тип на елементите на масив. Масивите също може да се разглеждат като допустим тип. Така можем да имаме масив от масиви.

Едномерен масив от цели числа декларираме с **int[]**. Ако желаем да декларираме масив от масиви от тип **int[]**, трябва всеки елемент да е от тип **int[]**, т.е. получаваме декларацията:

```
int[][] twoDimentionalArray;
```

Такива масиви ще наричаме **двумерни**, защото имат две измерениями или още **матрици** (терминът идва от математиката). Масиви с повече от едно измерение ще наричаме **многомерни**.

Аналогично можем да декларираме и **тримерни** масиви като добавим още едно измерение:

```
int[][][] threeDimentionalArray;
```

На теория няма ограничения за броя на размерностите на тип на масив, но в практиката масиви с повече от две размерности са рядко използвани, затова ще се спрем по-подробно на двумерните масиви.

Деклариране и заделяне на многомерен масив

Многомерните масиви се декларират по начин аналогичен на едномерните. Всяка размерност означаваме с квадратни скоби:

```
int[][] intMatrix;
float[][] floatMatrix;
String[][][] strCube;
```

Горният пример показва как да създадем двумерни и тримерни масиви. Всяка размерност отговаря на едни **[]**.

Памет за многомерни размери се заделя като се използва ключовата дума **new** и за всяка размерност в квадратни скоби се задава размера, който е необходим:

```
int[][] intMatrix = newint[3][4];
float[][] floatMatrix = newfloat[8][2];
```

```
String[][][] stringCube = new String[5][5][5];
```

В горния пример **intMatrix** е двумерен масив с 3 елемента от тип **int[]** и всеки от тези 3 елемента има размерност 4. Така представени, двумерните масиви изглеждат трудни за осмисляне. Затова може да ги разглеждаме като двумерни **матрици**, които имат редове и колони за размерности:

	0	1	2	3
0	1	2	6	3
1	9	0	7	1
3	2	8	5	4

Редовете и колоните се номерират с индекси от 0 до големината на съответната размерност минус едно. Ако един двумерен масив има размер **m** на **n**, той има **m*n** елемента.

Понякога можем да имаме неправоъгълни двумерни масиви, в които на всеки ред има различен брой колони.

Инициализация на многомерен масив

Инициализацията на многомерни масиви е аналогична на инициализацията на едномерните. Стойностите на елементите могат да се изброяват непосредствено след декларацията:

```
int[][] matrix = {
        {1, 2, 3, 4}, // row 0 values
        {5, 6, 7, 8}, // row 1 values
}
```

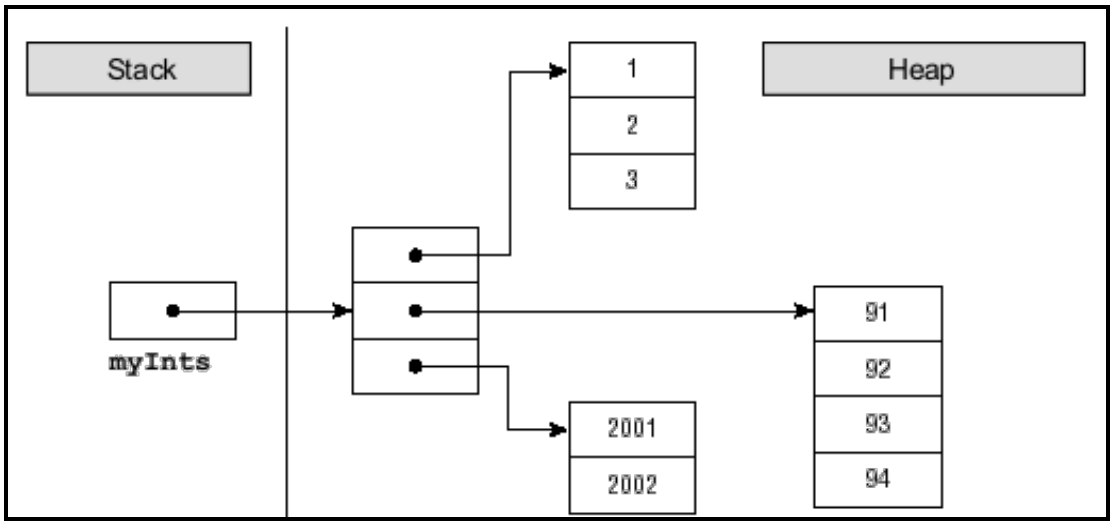
В горния пример инициализираме двумерен масив с цели числа с 2 реда и 4 колони. Във външните фигурни скоби се поставят елементите от първата размерност, т.е. редовете на двумерната матрица. Всеки ред представлява едномерен масив, който се инициализира по познат за нас начин.

Двумерните масиви и паметта

В паметта двумерните и многомерните масиви съхраняват стойностите си в динамичната памет като референция (указател) към област, съдържаща референции към други масиви. На практика всяка променлива от тип масив (едномерен или многомерен) представлява референция към място в динамичната памет, където се съхраняват елементите на масива. Ако масивът е двумерен, неговите елементи са съответно масиви и за тях се пазят референции към динамичната памет, където стоят съответните им елементи. За да си представим визуално това, нека разгледаме следния масив:

```
int[][] myInts = { {1, 2, 3}, {91, 92, 93, 94}, {2001, 2002} };
```

Този масив не е стандартна матрица, защото е с неправоъгълна форма. Той се състои от 3 реда, като всеки от тях има различен брой колони. Това в Java е позволено и след като бъде инициализиран, масивът се представя в паметта по следния начин:



Дължина на многомерен масив

Всяка размерност на многомерен масив има собствена дължина, която е достъпна по време на изпълнение на програмата. Нека разгледаме следния пример за двумерен масив:

```
int[][] matrix = {
  { 1, 2, 3, 4 },
  { 5, 6, 7, 8 },
};
```

Можем да извлечем броя на редовете на този двумерен масив чрез **matrix.length**. Това на практика е дължината на едномерния масив, съдържащ референциите към своите елементи (които са също масиви). Извличането на дължината на **i**-ия ред става с **matrix[i].length**.

Отпечатване на матрица – пример

Със следващия пример ще демонстрираме как можем да отпечатваме двумерни масиви на конзолата:

```
int[][] matrix = {
  { 1, 2, 3, 4 }, // row 0 values
  { 5, 6, 7, 8 }, // row 1 values
};
for (introw = 0; row < matrix.length; row++) {
          for (int col = 0; col < matrix[0].length; col++) {
            System.out.printf("%d ", matrix[row][col]);
           }
           System.out.println();
}
```

Първо декларираме и инициализираме масива, който искаме да обходим и да отпечатаме на конзолата. Масивът е двумерен и за това използваме един цикъл, който ще се движи по редовете и втори, вложен цикъл, който за всеки ред ще се движи по колоните на масива. За всяка итерация по подходящ начин извеждаме текущия елемент на масива като го достъпваме по неговите два индекса. В крайна сметка, ако изпълним горния програмен фрагмент, ще получим следния резултат:

```
1 2 3 4
5 6 7 8
```

Упражнения

1. Да се напише програма, която създава масив с 20 елемента от целочислен тип и инициализира всеки от елементите със стойност равна на индекса на елемента умножен по 5. Елементите на масива да се изведат на конзолата.
2. Да се напише програма, която чете два масива от конзолата и проверява дали са еднакви.
3. Да се напише програма, която сравнява два масива от тип **char** лексикографски (буква по буква) и проверява кой от двата е по-рано в лексикографската подредба.
4. Напишете програма, която намира максималната редица от еднакви елементи в масив. Пример: {2, 1, 1, 2, 3, 3, **2, 2, 2**, 1} → {2, 2, 2}.
5. Напишете програма, която намира максималната редица от нарастващи елементи в масив. Пример: {3, **2, 3, 4**, 2, 2, 4} → {2, 3, 4}.
6. Да се напише програма, която чете от конзолата две цели числа N и K, и масив от N елемента. Да се намерят тези K елемента, които имат максимална сума.

7. Да се напише програма, която намира последователност от числа в масив, които имат сума равна на число, въведено от конзолата (ако има такава). Пример: {4, 3, 1, **4, 2, 5**, 8}, S=11 → {4, 2, 5}.

8. Напишете програма, която създава следните квадратни матрици и ги извежда на конзолата във форматиран вид. Размерът на матриците се въвежда от конзолата. Пример за **(4,4)**:

a)

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

b)

1	8	9	16
2	7	10	15
3	6	11	14
4	5	12	13

c)

7	11	14	16
4	8	12	15
2	5	9	13
1	3	6	10

d)*

1	12	11	10
2	13	16	9
3	14	15	8
4	5	6	7

9. Да се напише програма, която създава правоъгълна матрица с размер **(n, m)**. Размерността и елементите на матрицата да се четат от конзолата. Да се намери подматрицата с размер **(3,3)**, която има максимална сума.

10. Бройни системи.

Какво представляват бройните системи?

Бройните системи са начин за представяне (записване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна **азбука**.

По време на различните етапи от развитието на човечеството, различни бройни системи са придобивали известност. Трябва да се отбележи, че днес най-широко разпространение е получила арабската бройна система. Тя използва цифрите 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, като своя азбука. (Интересен е фактът, че изписването на арабските цифри в днешно време се различава от представените по-горе десет цифри, но въпреки това, те пак се отнасят за същата бройна система т.е. десетичната).

Освен азбука, всяка бройна система има и **основа**. Основата е число, равно на броя различни цифри, използвани от системата за записване на числата в нея. Например арабската бройна система е десетична, защото има 10 цифри. За основа може да се избере произволно число, чиято абсолютна стойност трябва да бъде различна от 0 и 1. Тя може да бъде и реално или комплексно число със знак.

В практическо отношение, възниква въпросът: коя е най-добрата бройна система, която трябва да използваме? За да си отговорим на този въпрос, трябва да решим, как ще се представи по оптимален начин едно число като записване (т.е. брой на цифрите в числото) и брой на цифрите, които използва съответната бройна система т.е. нейната основа. По математически път, може да се докаже, че най-доброто съотношение между дължината на записа и броя на използваните цифри, се постига при основа на бройната система Неперовото число (e = 2,718281828), което е основата на естествените логаритми. Да се работи в система с тази основа, е изключително неудобно, защото това число не може да се представи като отношение на две цели числа. Това ни дава основание да заключим, че оптималната основа на бройната система е 2 или 3. Въпреки, че 3 е по-близо до Неперовото число, то е неподходящо за техническа реализация. Поради тази причина, двоичната бройна система, е единствената подходяща за практическа употреба и тя се използва в съвременните електронноизчислителни машини.

Позиционни бройни системи

Бройните системи се наричат **позиционни**, тогава, когато мястото (позицията) на цифрите има значение за стойността на числото. Това означава, че стойността на цифрата в числото не е строго определена и зависи от това на коя позиция се намира съответната цифра в дадено число. Например в числото 351 цифрата 1 има стойност 1, докато при числото 1024 тя има стойност 1000. Трябва да се отбележи, че основите на бройните системи се прилагат само при позиционните бройни системи. В позиционна бройна система числото$A_{(p)} = (a_{(n)}a_{(n-1)}...a_{(0)},a_{(-1)}a_{(-2)}...a_{(-k)})$ може де се представи във вида:

$$A_{(p)} = \sum_{m=n}^{-k} a_m T_m$$

В тази сума T_m има значение на теглови коефициент за m-тия разряд на числото. В повечето случаи обикновено$T_m=P^m$, което означава, че

$$A_{(p)} = \sum_{m=n}^{-k} a_m P^m$$

Образувано по горната сума, числото $A_{(p)}$е съставено съответно от цяла си част $(a_{(n)}a_{(n-1)}...a_{(0)})$ и от дробна си част $(a_{(-1)}a_{(-2)}...a_{(-k)})$, където всяко$a$ принадлежи на множеството от цели числа $M=\{0,1,2,..., p-1\}$. Лесно се вижда, че, при позиционните бройни системи стойността на всеки разряд е по-голяма от стойността на предходния разряд (съседния разряд отдясно, който е по-младши) с толкова пъти, колкото е основата на бройната система. Това обстоятелство, налага при събиране да прибавяме единица към левия (по-старшия) разряд, ако трябва да представим цифра в текущия разряд, която е по-голяма от основата. Системите с основи 2, 8, 10 и 16 са получили по-широко разпространение в изчислителната техника, и в следващата таблица е показано съответното представяне на числата от 0 до 15 в тях:

Двоична	**Осмична**	**Десетична**	**Шестнадесетична**
0000	**0**	**0**	**0**
0001	**1**	**1**	**1**

0010	**2**	**2**	**2**
0011	**3**	**3**	**3**
0100	**4**	**4**	**4**
0101	**5**	**5**	**5**
0110	**6**	**6**	**6**
0111	**7**	**7**	**7**
1000	**10**	**8**	**8**
1001	**11**	**9**	**9**
1010	**12**	**10**	**A**
1011	**13**	**11**	**B**
1100	**14**	**12**	**C**
1101	**15**	**13**	**D**
1110	**16**	**14**	**E**
1111	**17**	**15**	**F**

Непозиционни бройни системи

Освен позиционни, съществуват и непозиционни бройни системи, при които стойността на всяка цифра е постоянна и не зависи по никакъв начин от нейното място в числото. Като примери за такива бройни системи могат да се посочат съответно римската, гръцката, милетската и др. Като основен недостатък, на непозиционните бройни системи трябва да се посочи това, че чрез тях големите числа се представят неефективно. Заради този си недостатък те са получили по-ограничена употреба. Често това би могло да бъде източник на грешка при определяне на стойността на числата. Съвсем накратко ще разгледаме римската и гръцката бройни системи.

Римска бройна система

Римската бройна система използва следните символи за представяне на числата:

Римска цифра	**Десетична равностойност**
I	**1**
V	**5**
X	**10**

L	**50**
C	**100**
D	**500**
M	**1000**

Както вече споменахме, в тази бройна система позицията на цифрата не е от значение за стойността на числото, но за нейното определяне се прилагат следните правила:

1. Ако две последователно записани римски цифри, са записани така, че стойността на първата е по-голяма или равна на стойността на втората, то техните стойности се събират. Пример:
   Числото III=3, а числото MMD=2500.

2. Ако две последователно записани римски цифри, са в намаляващ ред на стойностите им, то техните стойности се изваждат. Пример:
   Числото IX=9, числото XML=1040, а числото MXXIV=1024.

Десетични числа

Числата представени в десетична бройна система, се задават в първичен вид т.е. вид удобен за възприемане от човека. Тази бройна система има за основа числото 10. Числата записани в нея са подредени по степените на числото 10. Младшият разряд (първият отдясно на ляво) на десетичните числа се използва за представяне на единиците (10^0=1), следващият за десетиците (10^1=10), следващият за стотиците (10^2=100) и т.н. Казано с други думи, всеки следващ разряд е десет пъти по-голям от предшестващия го разряд. Сумата от отделните разряди определя стойността на числото. За пример ще вземем числото 95031, което в десетична бройна система се представя като:

$$95031 = (\mathbf{9}x10^4) + (\mathbf{5}x10^3) + (\mathbf{0}x10^2) + (\mathbf{3}x10^1) + (\mathbf{1}x10^0)$$

Представено в този вид, числото 95031 е записано по естествен за човека начин, защото принципите на десетичната система са възприети като фундаментални за

хората. Много е важно да се отбележи, че тези подходи важат и за останалите бройни системи. Те имат същата логическа постановка, но тя е приложена за бройна система с друга основа. Последното твърдение, се отнася включително и за двоичната и шестнайсетината бройни системи, които ще разгледаме в детайли след малко.

Двоични числа

Числата представени в тази бройна система, се задават във вторичен вид т.е. вид удобен за възприемане от изчислителната машина. Този вид е малко по-трудно разбираем за човека. За представянето на двоичните числа, се използва двоичната бройна система, която има за основа числото 2. Числата записани в нея са подредени по степените на двойката. За тяхното представяне, се използват само цифрите 0 и 1.

Прието е, когато едно число се записва в бройна система, различна от десетичната, във вид на индекс в долната му част да се отразява, коя бройна система е използвана за представянето му. Например със записа $\mathbf{1110_{(2)}}$ означаваме число в двоична бройна система. Ако не бъде указана изрично, бройната система се приема, че е десетична. Числото се произнася, като се прочетат последователно неговите цифри, започвайки от ляво на дясно (т.е. прочитаме го от старшия към младия разряд "бит").

Както и при десетичните числа, гледано от дясно наляво, всяко двоично число изразява степените на числото 2 в съответната последователност. На младшата позиция в двоично число съответства нулевата степен ($2^0=1$), на втората позиция съответства първа степен ($2^1=2$), на третата позиция съответства втора степен ($2^2=4$) и т.н. Ако числото е 8-битово, степените достигат до седма ($2^7=128$). Ако числото е 16-битово, степените достигат до петнадесета ($2^{15}=32768$). Чрез 8 двоични цифри (0 или 1) могат да се представят общо 256 числа, защото $2^8=256$. Чрез 16 двоични цифри могат да се представят общо 65536 числа, защото $2^{16}=65536$.

Нека даден един пример за числа в двоична бройна система. Да вземем десетичното число **148**. То е съставено от три цифри: **1, 4** и **8**, и съответства на следното двоично число:

$\mathbf{10010100}_{(2)}$

$\mathbf{148} = (\mathbf{1}x2^7) + (\mathbf{1}x2^4) + (\mathbf{1}x2^2)$

	Ако едно двоично число завършва на 0, то е четно, а ако завършва на 1, то е нечетно.

Преминаване от двоична в десетична бройна система

При преминаване от двоична в десетична бройна система, се извършва преобразуване на двоичното число в десетично. Всяко число може да се преобразува от една бройна система в друга, като за целта се извършат последователност от действия, които са възможни и в двете бройни системи. Както вече споменахме, числата записани в двоична бройна система се състоят от двоични цифри, които са подредени по степените на двойката. Нека да вземем за пример числото $\mathbf{11001}_{(2)}$. Преобразуването му в десетично се извършва чрез пресмятането на следната сума:

$\mathbf{11001}_{(2)} = \mathbf{1}x2^4 + \mathbf{1}x2^3 + \mathbf{0}x2^2 + \mathbf{0}x2^1 + \mathbf{1}x2^0 =$

$= \mathbf{16}_{(10)} + \mathbf{8}_{(10)} + \mathbf{1}_{(10)} = \mathbf{25}_{(10)}$

От това следва, че $\mathbf{11001}_{(2)} = \mathbf{25}_{(10)}$

С други думи, всяка една двоична цифра се умножава по 2 на степен, позицията, на която се намира (в двоичното число). Накрая се извършва събиране, на числата, получени за всяка от двоичните цифри, за да се получи десетичната равностойност на двоичното число.

Съществува и още един начин за преобразуване, който е известен като схема на Хорнер. При тази схема се извършва умножение на най-лявата цифра по две и събиране със съседната ѝ вдясно. Този резултат се умножава по две и се прибавя

следващата съседна цифра от числото (цифрата вдясно). Това продължава до изчерпване на всички цифри в числото, като последната цифра от числото се добавя без умножаване. Ето един пример:

$\mathbf{1001}_{(2)}$ = ((1.2+0).2+0).2+1 = 2.2.2+1 = **9**

Преминаване от десетична към двоична бройна система

При преминаване от десетична в двоична бройна система, се извършва преобразуване на десетичното число в двоично. За целите на преобразуването се извършва делене на две с остатък. Така се получават частно и остатък, който се отделя.

Отново ще вземем за пример числото 148. То се дели целочислено на основата, към която ще преобразуваме (в примера тя е 2). След това, от остатъците получени при деленето (те са само нули и единици), се записва преобразуваното число. Деленето продължава, докато получим частно нула. Ето пример:

148:2=74 имаме остатък 0;

74:2=37 имаме остатък 0;

37:2=18 имаме остатък 1;

18:2=9 имаме остатък 0;

9:2=4 имаме остатък 1;

4:2=2 имаме остатък 0;

2:2=1 имаме остатък 0;

1:2=0 имаме остатък 1;

След като вече семе извършили деленето, записваме стойностите на остатъците в ред, обратен на тяхното получаване, както следва:

10010100

т.е. $148_{(10)} = 10010100_{(2)}$

Шестнайсетични числа

При тези числа имаме за основа на бройната система числото 16, което налага да бъдат използвани 16 знака (цифри) за представянето на всички възможни стойности от 0 до 15 включително. Както вече беше показано в една от таблиците в предходните точки, за представянето на шестнайсетичните числа се използват числата от 0 до 9 и латинските букви от A до F. Всяка от тях има съответната стойност:

A=10, B=11, C=12, D=13, E=14, F=15

Като примери за шестнайсетични числа могат да бъдат посочени съответно, D2, 1F2 F1, D1E и др.

Преминаването към десетична система става като се умножи по 16^0 стойността на най-дясната цифра, по 16^1 следващата вляво, по 16^2 следващата вляво и т.н. и накрая се съберат. Например:

$D1E_{(16)} = E*16^0 + 1*16^1 + D*16^2 = 14*1 + 1*16 + 13*256 = 3358_{(10)}$.

Преминаването от десетична към шестнайсетична бройна система става като се дели десетичното число на 16 и се вземат остатъчите в обратен ред. Например:

3358 / 16 = 209 + остатък 14 (E)

209 / 16 = 13 + остатък 1 (1)

13 / 16 = 0 + остатък 13 (D)

Взимаме остатъците в обратен ред и получаваме числото $D1E_{(16)}$.

Бързо преминаване от двоични към шестнайсетични числа

Бързото преобразуване, от двоични в шестнайсетични числа се извършва бързо и лесно, чрез разделяне на двоичното число на групи от по четири бита (разделяне на полубайтове). Ако броят на цифрите в числото не е кратен на четири, то се

добавят водещи нули в старшите разреди. След разделянето и евентуалното добавяне на нули, се заместват всички получени групи със съответстващите им цифри. Ето един пример:

Нека да ни е дадено следното число: $1110011110_{(2)}$.

1. Разделяме го на полубайтове и добавяме водещи нули
   Пример: 0011 1001 1110.

2. Заместваме всеки полубайт със съответната шестнайсетична цифра и така получаваме $39E_{(16)}$.

Следователно $1110011110_{(2)} = 39E_{(16)}$.

Представяне на числата

За съхраняване на данните в оперативната памет на електронноизчислителните машини, се използва двоичен код. В зависимост от това какви данни съхраняваме (символи, цели или реални числа с цяла и дробна част) информацията се представя по различен начин. Този начин се определя от типа на данните.

Дори и програмистът на език от високо ниво трябва да знае, какъв вид имат данните разположени в оперативната памет на машината. Това се отнася, и за случаите, когато данните се намират на външен носител, защото при обработката им те се разполагат в оперативната памет.

В тази глава са разгледани начините за представяне и обработка на различни типове данни. Най-общо те се основават на понятията бит, байт и машинна дума.

Бит е една двоична единица от информация, със стойност 0 или 1.

Информацията в паметта се групира в последователности от 8 бита, които образуват един **байт.**

За да бъдат обработени от аритметичното устройство, данните се представят в паметта от определен брой байтове (2, 4 или 8), които образуват машинната

дума. Това са концепции, които всеки програмист трябва задължително да знае и разбира.

Представяне на цели числа в паметта

Едно от нещата, на които до сега на обърнахме внимание е знакът на числата. Представянето на цели числата в паметта на компютъра, може да се извърши по два начина: със знак или без знак. Когато числата се представят със знак се въвежда знаков разред. Той е най-старшият разред и има стойност 1 за отрицателните числа и 0 за положителните. Останалите разреди са информационни и отразяват (съдържат) стойността числото. В случая на числа без знак всички битове се използват за записване на стойността на числото.

Цели числа без знак

За целите числа без знак се заделят по 1, 2, 4 или 8 байта от паметта. В зависимост, от броя на байтовете използвани при представянето на едно число, се образуват обхвати на представяне с различна големина. Посредством*n* на брой бита могат да се представят цели числа без знак в обхвата $[0, 2^n-1]$. Следващата таблица, показва обхвата от стойности на целите числа без знак:

Брой байтове за представяне на числото в паметта	**Обхват**	
	Запис чрез порядък	**Обикновен запис**
1	**$0 \div 2^8-1$**	**0 ÷ 255**
2	**$0 \div 2^{16}-1$**	**0 ÷ 65 535**
4	**$0 \div 2^{32}-1$**	**0 ÷ 4 294 967 295**
8	**$0 \div 2^{64}-1$**	**0 ÷ 9 223 372 036 854 775 807**

Ще покажем пример при еднобайтово и двубайтово представяне на числото 158, което се записва в двоичен вид като $10011110_{(2)}$:

1. Представяне с 1 байт:

1	0	0	1	1	1	1	0

2. Представяне с 2 байта:

0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0

Представяне на отрицателни числа

За отрицателните числа се заделят по един, два или четири байта от паметта на компютъра, като най-старшият разред има значение на знаков и ни носи информация за знака на числото. Както вече споменахме, когато знаковият бит има стойност 1 числото е отрицателно, а в противен случай – положително. Следващата таблица, показва обхвата от стойности на целите числа със знак:

Брой байтове за представяне на числото в паметта	**Обхват**	
	Запис чрез порядък	**Обикновен запис**
1	**$-2^7 \div 2^7-1$**	**-128 ÷ 127**
2	**$-2^{15} \div 2^{15}-1$**	**-32 768 ÷ 32 767**
4	**$-2^{31} \div 2^{31}-1$**	**-2 147 483 648 ÷ 2 147 483 647**
8	**$-2^{63} \div 2^{63}-1$**	**-9 223 372 036 854 775 808 ÷ 9 223 372 036 854 775 807**

За кодирането на отрицателните числа, се използват прав, обратен и допълнителен код. И при трите представяния целите числа със знак са в границите: $[-2^{n-1}, 2^{n-1}-1]$. Положителните числа винаги се представят по един и същи начин и за тях правият, обратният и допълнителният код съвпадат.

Прав код: Правият код е най-простото представяне на числото. Старшият бит е знаков, а в оставащите битове е записана абсолютната стойност на числото. Ето няколко примера:

Числото 3 в прав код се представя в осембитово число като 00000011.

Числото -3 в прав код се представя в осембитово число като 10000011.

Обратен код: Получава се от правия код на числото, чрез инвертиране (заместване на всички нули с единици и единици с нули). Този код не е никак удобен за извършването на аритметичните действия събиране и изваждане, защото се изпълнява по различен начин, когато се налага изваждане на числа.

Освен това се налага знаковите битове да се обработват отделно от информационните. Този недостатък се избягва с употребата на допълнителен код, при който вместо изваждане се извършва събиране с отрицателно число. Последното е представено чрез неговото допълнение т.е. разликата между 2^n и самото число. Пример:

Числото -127 в прав код се представя като 1 1111111, а в обратен код като 1 0000000.

Числото 3 в прав код се представя като 0 0000011, а в обратен код има вида 0 1111100.

Допълнителен код: Допълнителният код е число в обратен код, към което е прибавена (чрез събиране) единица. Пример:

Числото -127 представено в допълнителен код има вида 1 0000001.

Двоично-десетичен код: Известен е още като BCD код (Binary Coded Decimal). При този код в един байт се записват по две десетични цифри. Това се постига, като чрез всеки полубайт се кодира една десетична цифра. Числа представени чрез този код могат да се пакетират т.е. да се представят в пакетиран формат. Ако представим една десетична цифра в един байт се получава непакетиран формат.

Съвременните микропроцесори използват един или няколко от разгледаните кодове за представяне на отрицателните числа, като най-разпространеният начин е представянето в допълнителен код.

Стриктен режим на изчисленията с плаваща запетая

Изчисленията с плаваща запетая могат да се изпълняват и в стриктен режим. Стриктната аритметика при числата с плаваща запетая следва строги правила за операциите, които гарантират, че ще получим един и същ резултат от изчисленията при изпълнение на програмата на различни версии на виртуалната машина.

Ако се налага да гарантирате побитова еднаквост на резултата във всяка реализация на виртуалната машина, трябва да ползвате модификатора **strictfp**. Той може да се приложи към клас, интерфейс или метод.

Числа с фиксирана запетая

В някои езици за програмиране съществуват и числа с **фиксирана запетая**, например типът **decimal** в C#, типът **money** в SQL Server и типът **number(10,2)** в Oracle. В Java за съжаление няма такъв примитивен тип.

Ако се нуждаем от точни изчисления (например за счетоводни или финансови цели), можем да ползваме класа **BigDecimal**, който не губи точност при пресмятанията, но за сметка на това работи чувствително по-бавно от **float** и **double**. Ето как изглежда в нов вариант нашата програма, която правеше грешки при сумиране на числа:

```
import java.math.BigDecimal;


publicclass Precision {
        publicstaticvoid main(String[] args) {
                double sum = 0.0d;
                BigDecimal bdValue = new BigDecimal("0.1");
                BigDecimal bdSum = new BigDecimal("0.0");
                for(int i=1; i<=10; i++) {
                        sum += 0.1d;
                        bdSum = bdSum.add(bdValue);
                }
                System.out.println("Double sum is: " + sum);
                System.out.println("BigDecimal sum is: " + bdSum);
        }
}
```

По време на нейното изпълнение, отново добавяме в цикъл стойността една десета в променливата sum от тип double и обектната променливата от тип **BigDecimal**. След изпълнението на програмата ще получим:

```
Double sum is: 0.9999999999999999
BigDecimal sum is: 1.0
```

Упражнения

1. Превърнете числата 151, 35, 43, 251 и -0,41 в двоична бройна система.
2. Превърнете числото $\mathbf{1111010110011110_{(2)}}$в шестнадесетична и десетична бройна система.
3. Превърнете шестнайсетичните числа 2A3E, FA, FFFF, 5A0E9 в двоична и десетична бройна система.
4. Да се напише програма, която преобразува десетично число в двоично.
5. Да се напише програма, която преобразува двоично число в десетично.
6. Да се напише програма, която преобразува десетично число в шестнадесетично.

11. Методи.

Какво е "метод"?
Метод (method) е съставна част от програмата, която решава даден проблем.

В методите се извършва цялата обработка на данни, която програмата трябва да направи, за да реши поставената задача. Те са мястото, където се извършва реалната работа. Затова можем да ги приемем като строителен блок на програмата. Съответно, имайки множество от простички блокчета – отделни методи, можем да създаваме големи програми, с които да решим сложни проблеми. Ето как изглежда един метод за намиране лице на правоъгълник например:

```
publicstaticdouble getRectagnleArea(
                double width, double height) {
        double area = width * height;
        return area;
}
```

Защо да използваме методи?
Има много причини, които ни карат да използваме методи. Ще разгледаме някои от тях и с времето ще се убедите, че методите са нещо, без което не можем, ако искаме да програмираме сериозно.

По-добро структуриране и по-добра четимост

При създаването на една програма, е добра практика да използваме методи, за да я направим по-добре структурирана и по-лесно четима не само за нас, но и за други хора.

Довод за това е, че за времето, през което съществува една програма, само 20% от усилията, които се заделят за нея, се състоят в създаване и тестване на кода. Останалата част е за поддръжка и добавяне на нови функционалности към началната версия. В повечето случаи, след като веднъж кодът е написан, той не

се поддържа и модифицира само от създателя му, но и от други програмисти. Затова е добре той да е добре структуриран и лесно четим.

Избягване на повторението на код

Друга причина, заради която е добре да използваме методи е, че по този начин избягваме повторението на код. Това е пряко свързано със следващата точка – преизползване на кода.

Деклариране на метод наричаме регистрирането на метода, за да бъде разпознаван в останалата част на Java-света.

Имплементация(създаване) на метода, е реалното написване на кода, който решава конкретната задача, заради която се създава метода. Този код се съдържа в самия метод.

Извикване е процесът на стартиране на изпълнението, на вече декларирания и създаден метод, от друго място на програмата, където трябва да се реши проблемът, който нашият метод решава.

Деклариране на собствен метод

Преди да се запознаем как можем да декларираме метод, трябва да знаем къде е позволено да го направим.

Къде е позволено да декларираме метод

Въпреки, че формално все още не сме запознати как се декларира клас, от примерите, които сме разглеждали до сега в предходните глави, знаем, че всеки клас има отваряща и затваряща фигурни скоби – "**{**" и "**}**", между които пишем програмния код. Повече подробности за това, ще научим в главата "Дефиниране на класове", но го споменаваме тук, тъй като един метод може да съществува само ако е деклариран **между** отварящата и затварящата скоби на даден клас – "**{**" и "**}**". Също така методът, трябва да бъде деклариран **извън** имплементацията на друг метод (за това малко по-късно).

Можем да декларираме метод единствено в рамките на даден клас – между отварящата "{" и затварящата "}" му скоби.

Най-очевидния пример за това е методът **main()** – винаги го декларираме между отварящата и затварящата скоба на нашия клас, нали?

HelloJava.java

```
publicclass HelloJava {
        publicstaticvoid main(String[] args) {
                System.out.println("Hello Java!");
        }
}
```

Декларация на метод

Декларирането на метода, представлява **регистриране** на метода в нашата програма. То става по следния начин:

```
[public] [static] <return_type><method_name>([<param_list>])
```

Задължителните елементи в декларацията на един метод са:

- Тип на връщаната от метода стойност – **<return_type>**.
- Име на метода – **<method_name>**.
- Списък с параметри на метода – **<param_list>** – съществува само ако метода има нужда от тях в процеса на работата си.

За онагледяване на това, можем да погледнем **main()** метода в примера **HelloJava** от предходната секция:

```
publicstaticvoid main(String[] args)
```

При него, типа на връщаната стойност е **void** (т.е. метода не връща резултат), името му е **main**, следвано от кръгли скоби, в които има списък с параметри, състоящ се от един параметър – масивът **String[] args**.

Последователността, в която трябва да се поставят отделните елементи от декларацията на метода е строго определена. Винаги на първо място е типът на връщаната стойност **<return_type>**, следвана от името на метода **<method_name>**и накрая, списък с параметри **<param_list>** ограден с кръгли скоби – "**(**" и "**)**".

При деклариране на метод, спазвайте последователността, в която се описват основните му характеристики: първо тип на връщана стойност, след това име на метода и накрая списък от параметри ограден с кръгли скоби.

Списъкът от параметри може да е празен (тогава просто пишем "**()**" след името на метода). Дори методът да няма параметри, кръглите скоби трябва да присъстват в декларацията му.

Кръглите скоби – "(" и ")", винаги следват името на метода, независимо дали той е с или без параметри.

За момента, ще пропуснем разглеждането какво е**<return_type>** и само ще кажем, че на това място трябва да стои ключовата дума **void**, която указва, че методът не връща никаква стойност. По-късно в тази глава, ще видим какво представлява и какво можем да поставим на нейно място.

Име на метода

Всеки метод, решава някаква подзадачка от цялостния проблем, с който се занимава програмата ни. Когато създаваме програмата и стигнем до подпроблема, който този метод решава, ние извикаме (стартираме) метода, използвайки името му.

В примера показан по-долу, името на метода е **printLogo**:

```
publicstaticvoid printLogo() {
        System.out.println("Sun Microsystems");
        System.out.println("www.sun.com");
}
```

Правила за създаване на име на метод

Добре е, когато декларираме името на метода, да спазваме правилата за именуване на методи, препоръчани ни от Sun:

- Името на метода трябва да започва с малка буква.
- Трябва да се прилага правилото **camelCase**, т.е. всяка нова дума, която се долепя в задната част на името на метода, започва с главна буква.
- Имената на методите е добре да бъдат съставени от глагол или от глагол и съществително име.

Нека отбележим, че тези правила не са задължителни, а препоръчителни. Но принципно, ако искаме форматирането на кода ни да е като на всички Java-програмисти по света е добре да спазваме конвенциите на Sun.

Ето няколко примера:

```
print
getName
playMusic
setUserName
```

Освен това, името на метода трябва да описва неговата цел. Идеята е, ако човек, който не е запознат с програмата ни, прочете името на метода, да добие представа какво прави този метод, без да се налага да разглежда кода му.

При определяне на името на метод се препоръчва да се спазват следните правила:

- **Името на метода трябва да описва неговата цел.**

	- **Името на метода трябва да започва с малка буква.** - **Трябва да се прилага правилото camelCase.** - **Името на метода трябва да е съставено от глагол или от двойка - глагол и съществително име.**

Тяло на метод

Тяло на метод наричаме програмния код, който се намира между фигурните скоби "**{**" и "**}**", следващи непосредствено декларацията на метода.

```
publicstatic<return_type><method_name>(<parameters_list>) {
        }
```

Реалната работа, която методът извършва, се намира именно в тялото на метода. В него трябва да бъде описан алгоритъмът, по който методът решава поставения проблем.

Пример, за тяло на метод сме виждали много пъти, но сега ще изложим отново един:

```
publicstaticvoid printLogo() {
        System.out.println("Sun Microsystems");
        System.out.println("www.sun.com");
}
```

Преди да приключим със секцията за тяло на метод, трябва отново да обърнем внимание на едно от правилата, къде може да се декларира метод:

	**Метод НЕ може да бъде деклариран в тялото на друг метод.**

Поведение на метода в зависимост от входните данни

Когато декларираме метод с параметри, целта ни е всеки път, когато извикваме този метод, работата му да се променя в зависимост от входните данни. С други

думи, алгоритъмът, който ще опишем в метода, ще бъде един, но крайният резултат ще бъде различен, в зависимост каква информация сме подали на метода чрез списъка от параметри.

	**Когато методът ни приема параметри, поведението му, зависи от тях.**

Списък с много параметри

До сега разглеждахме примери, в които методите имат списък от параметри, който се състои от един единствен параметър. Когато декларираме метод обаче, той може да бъде има толкова параметри, колкото са му необходими.

Например, когато търсим по-голямото от две числа, ние подаваме два параметъра:

```
publicstaticvoid printMax(float number1, float number2) {
        float max = number1;
        if (number2 > number1) {
                max = number2;
        }
        System.out.println("Maximal number: " + max);
}
```

Особеност при декларацията на списък с много параметри

Когато в списъка с параметри декларираме повече от един параметър от един и същ тип, трябва да знаем, че не можем да използваме съкратения запис за деклариране на променливи от един и същ тип, както е позволено в самото тяло на метода:

```
float var1, var2;
```

Винаги трябва да указваме типа на параметъра в списъка с параметри на метода, независимо че някой от съседните му параметри е от същия тип.

Например, тази декларация на метод е неправилна:

```
publicstaticvoid printMax(float var1, var2)
```

Съответно, правилният начин е:

```
publicstaticvoid printMax(float var1, float var2)
```

Извикване на метод с параметри

Извикването на метод с много параметри става по същия начин, по който извиквахме на метод без параметри. Разликата е, че между кръглите скоби, след името на метода, поставяме стойности. Тези стойности ще бъдат присвоени на съответните параметри от декларацията на метода и при изпълнението си, метода ще работи с тях.

Ето няколко примера на извикване на методи с параметри:

```
printSign(-5);
printSign(balance);
printMax(100, 200);
```

Разлика между параметри и аргументи на метод

Преди да продължим, трябва да направим едно разграничение между наименованията на параметрите в списъка от параметри в декларацията на метода и стойностите, които подаваме при извикването на метода.

За по-голяма яснота, при декларирането на метода, елементите на списъка от параметрите му, ще наричаме **параметри** (някъде в литературата могат да се срещнат също като "формални параметри").

По време на извикване на метода, **стойностите**, които подаваме на метода, наричаме **аргументи** (някъде могат да се срещнат под понятието "фактически параметри").

С други думи, елементите на списъка от параметри **var1** и **var2**, наричаме параметри:

```
publicstaticvoid printMax(float var1, float var2)
```

Съответно стойностите, при извикването на метода **-23.5** и **100**, наричаме аргументи:

```
printMax(100, -23.5);
```

Подаване на аргументи от примитивен тип

Както току-що научихме, когато в Java подадем като аргумент на метод дадена променлива, стойността й се **копира** в параметъра от декларацията на метода. След това, копието ще бъде използвано в тялото на метода. Има, обаче, една особеност.

Когато съответният параметър от декларацията на метода е от **примитивен тип**, това практически не оказва никакво влияние на кода след извикването на метода.

Например, ако имаме следния метод:

```
publicstaticvoid printNumber(int numberParam) {
        numberParam = 5;
        System.out.println("in printNumber() method, after the "
                        + "modification, numberParam is: " +
numberParam);
}
```

Извиквайки го от метода **main()**:

```
publicstaticvoid main(String[] args) {
```

```
        int numberArg = 3;
        printNumber(numberArg);
        System.out.println("in the main() method number is: " +
                numberArg);
}
```

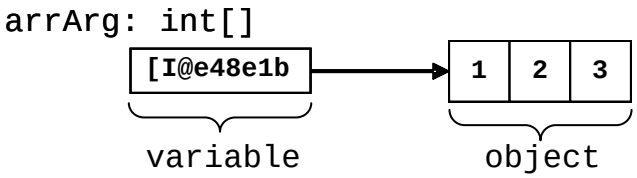
Стойността **3** на променливата **numberArg**, се копира в параметъра **numberParam**. След като бъде извикан методът **printNumber()**, на параметъра **numberParam** се присвоява стойността **5**. Това не рефлектира върху стойността на променливата **numberArg**, тъй като при извикването на метода, в **numberParam** се пази **копие** на стойността на подадения аргумент. Затова, методът **printNumber()** отпечатва числото **5**. Съответно, след извикването на метода **printNumber()**, в метода **main()** отпечатваме стойността на променливата **numberArg** и виждаме, че тя не е променена. Ето и изходът от изпълнението на горния код:

```
in printNumber() method, after the modification numberParam is:5
in the main() method number is: 3
```

Подаване на аргументи от референтен тип

Когато трябва да декларираме (и съответно извикаме) метод, чийто параметри са от **референтен тип** (например масиви), трябва да бъдем много внимателни.

Преди да обясним защо, нека припомним нещо от главата "Масиви". Масивът, като всеки референтен тип, се състои от променлива (референция) и стойност – реалната информация в паметта на компютъра (нека я наречем **обект**). Съответно в нашия случай обектът представлява реалният масив от елементи. Променливата пази адреса на обекта (елементите на масива) в паметта:



Когато оперираме с масиви, винаги го правим чрез променливата, с която сме ги декларирали. Така е и с всеки референтен тип. Следователно, когато подаваме аргумент от референтен тип, **стойността**, която е записана в променливата-аргумент, се **копира** в променливата, която е параметър в списъка от параметри на метода. Но какво става с обекта (реалния масив от елементи)? Копира ли се и той или не?

За да бъде по-нагледно обяснението, нека използваме следния пример: имаме метод **modifyArr()**, който модифицира първия елемент на подаден му като параметър масив, като го реинициализира със стойност **5** и след това отпечатва елементите на масива, оградени в квадратни скоби и разделени със запетайки:

```
publicstaticvoid modifyArr(int[] arrParam) {
        arrParam[0] = 5;

        System.out.print("In modifyArr() the param is: ");
        System.out.println(Arrays.toString(arrParam));
}
```

Съответно, декларираме и метод **main()**, от който извикваме новосъздадения метод **modifyArr()**:

```
publicstaticvoid main(String[] args) {
        int[] arrArg = newint[] { 1, 2, 3 };
        System.out.print("Before modifyArr() the argument is: ");
        System.out.println(Arrays.toString(arrArg));
        modifyArr(arrArg);
```

```
        System.out.print("After modifyArr() the argument is: ");
        System.out.println(Arrays.toString(arrArg));
}
```

Какъв ще е резултатът от изпълнението на този код? Нека погледнем:

```
Before modifyArr() the argument is: [1, 2, 3]
In modifyArr() the param is: [5, 2, 3]
After modifyArr() the argument is: [5, 2, 3]
```

Забелязваме, че след изпълнението на метода **modifyArr()**, масивът към който променливата **arrArg** пази референция, не е **[1,2,3]**, а е **[5,2,3]**. Какво значи това?

Причината за този резултат е, че при подаването на аргумент от референтен тип, се копира единствено стойността на променливата, която пази референция към обекта, но **не се прави копие на самия обект**.

При подаване на аргументи от референтен тип се копира само стойността на променливата, която пази референция към обекта в паметта, но не и самият обект.

Метод с променлив брой аргументи (var-args)

До момента, разглеждахме деклариране на методи, при което декларираме списък от параметри на метода, при който, когато извикваме нашия метод, аргументите, които подаваме трябва да са същият брой, какъвто е броят на параметрите в декларацията му.

Сега ще разгледаме деклариране на методи, която позволява **по време на извикване на метода**, броят на аргументите, които биват подавани, да е различен, в зависимост от нуждите на извикващия код.

Нека вземем примера, който разгледахме по-горе, в който пресмятаме сумата, която заплащаме на продавача в книжарницата след като сме си избрали книги.

В него, като параметър на метода подавахме масив от тип **double[]**, в който се съхраняват цените на избраните от нас книги:

```
publicstaticvoid printTotalAmountForBooks(double[] prices) {
        double totalAmount = 0;


        for (double singleBookPrice : prices) {
                totalAmount += singleBookPrice;
        }
        System.out.println("The total amount of all books is: " +
                totalAmount);
}
```

Така дефиниран, този метод предполага, че **винаги** преди да го извикаме, ще създадем масив с числа от тип **double** и ще го инициализираме с някакви стойности.

След създаването на Java 5.0, е възможно, когато трябва да подадем някакъв списък от стойности от **един и същ тип** на даден метод, вместо да го правим като подаваме масив, който съдържа тези стойности, да ги подадем на метода при извикването му, като аргументи, разделени със запетая.

Например, в нашия случай с книгите, вместо да създаваме масив, специално заради извикването на този метод:

```
double[] prices = newdouble[] { 3, 2.5 };
printTotalAmountForBooks(prices);
```

Можем директно да подадем списъка с цените на книгите, като аргументи на метода:

```
printTotalAmountForBooks(3, 2.5);
printTotalAmountForBooks(3, 5.1, 10, 4.5);
```

Този тип извикване на метода обаче е възможно само ако сме декларирали метода си, като метод, който приема променлив брой аргументи (var-args).

Деклариране на метод с връщана стойност

Ако погледнем отново как декларираме метод:

```
publicstatic<return_type><method_name>(<parameters_list>)
```

Ще си припомним, че когато обяснявахме за това, казахме, че на мястото на **<return type>** поставяме **void**. Сега ще разширим дефиницията, като кажем, че на това място може да стои не само **void**, но и произволен тип – примитивен (**int**, **float**, **double**, …) или референтен (например **String** или масив), в зависимост от това, какъв тип е резултатът от изпълнението на метода.

Например, ако вземем примера с метод, който изчислява лице на квадрат, вместо да отпечатваме стойността в конзолата, методът може да я върне като резултат. Ето как би изглеждала декларацията на метода:

```
publicstaticdouble calcSquareSurface(double sideLength)
```

Виждаме, че резултатът от пресмятането на лицето е от тип **double**.

Употреба на връщаната стойност

Когато методът бъде изпълнен и върне стойност, можем да си представяме, че Java поставя тази стойност на мястото, където е било извикването на метода и продължава работа с нея. Съответно, тази върната стойност, можем да използваме от извикващия метод с различни цели.

Присвояване на променлива

Може да присвоим резултата от изпълнението на метода, на променлива от подходящ тип:

```
String companyLogo = getCompanyLogo();
```

Употреба в изрази

След като един метод върне резултат, този резултат, може да го използваме в изрази.

Например, за да намерим общата цена трябва да получим единичната такава и да умножим по количеството:

```
float totalPrice = getSinglePrice() * quantity;
```

Подаване като стойност в списък от параметри на друг метод

Можем да подадем резултата от работата на един метод, като стойност в списъка от параметри на друг метод:

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

В този пример, отначало извикваме метода **getCompanyLogo()**, подавайки го като аргумент на метода **println()**. След като методът **getCompanyLogo()** бъде изпълнен, той ще върне резултат, например – "**Sun Microsystems**". Тогава Java ще "подмени" извикването на метода, с резултата, който е върнат от изпълнението му и можем да приемем, че в кода имаме:

```
System.out.println("Sun Microsystems");
```

Тип на връщаната стойност

Както казахме малко по-рано, резултатът, който връща един метод може да е от всякакъв тип – **int**, **String**, масив и т.н. Когато обаче, като тип на връщаната стойност бъде употребена ключовата дума **void**, с това означаваме, че методът не връща никаква стойност.

Операторът return

За да накараме един метод да връща стойност, трябва в тялото му, да използваме ключовата дума **return**, следвана от резултата на метода:

```
public static <return_type><method_name>(<parameters_list>) {
```

```
        return <method's_result>;
}
```

Съответно **<method's_result>**, е от тип **<return_type>**. Например:

```
publicstaticint multiply(int number1, int number2) {
        int result = number1 * number2;
        return result;
}
```

В този метод, след умножението, благодарение на **return**, методът ще върне резултата от изпълнението на метода – целочислената променлива **result.**

Валидация на данни – пример

В тази задача, трябва да напишем програма, която пита потребителя колко е часът (с извеждане на въпроса "**What time is it?**"). След това потребителят, трябва да въведе две числа, съответно за час и минути. Ако въведените данни представляват валидно време, програмата, трябва да изведе съобщението "**The time is HH:mm now.**", където с **HH** съответно сме означили часа, а с **mm** – минутите. Ако въведените час или минути не са валидни, програмата трябва да изведе съобщението "**Incorrect time!**".

След като прочитаме условието на задачата внимателно, стигаме до извода, че решението на задачата може да се разбие на следните подзадачи:

- Получаване на входа за час и минути.
- Проверка на валидността на входните данни.
- Извеждаме съобщение за грешка или валидно време.

Знаем, че обработката на входа и извеждането на изхода няма да бъдат проблем за нас, затова решаваме да разрешим проблема с валидността на входните данни, т.е. валидността на числата за часове и минути. Знаем, че часовете варират от 0 до 23 включително, а минутите съответно от 0 до 59 включително. Тъй като данните (часове и минути) не са еднородни решаваме да създадем два отделни

метода, единият от които проверява валидността на часовете, а другия – на минутите.

Ето едно примерно решение:

DataValidation.java

```
import java.util.Scanner;

publicclass DataValidation {

        publicstaticvoid main(String[] args) {
                Scanner input = new Scanner(System.in);
                System.out.println("What time is it?");

                System.out.print("Hours: ");
                int hours = input.nextInt();

                System.out.print("Minutes: ");
                int minutes = input.nextInt();

                boolean isValidTime =
                        validateHours(hours) &&validateMinutes(minutes);
                if (isValidTime) {
                        System.out.printf(
                                "The time is %d:%d now.%n", hours,
minutes);
                } else {
                        System.out.println("Incorrect time!");
```

```
        }
    }

    publicstaticboolean validateHours(int hours) {
        boolean result = (hours >= 0) && (hours < 24);
        return result;
    }

    publicstaticboolean validateMinutes(int minutes) {
        boolean result = (minutes >= 0) && (minutes <= 59);
        return result;
    }
}
```

Методът, който проверява часовете, го кръщаваме **validateHours()**, като той приема едно число от тип **int**, за часовете и връща резултат от тип **boolean**, т.е. **true** ако въведеното число е валиден час и **false** в противен случай:

```
publicstaticboolean validateHours(int hours) {
    boolean result = (hours >= 0) && (hours < 24);
    return result;
}
```

По подобен начин, декларираме метод, който проверява валидността на минутите. Наричаме го **validateMinutes()**, като приема като списък от един параметър цяло число, за минути и има тип на връщана стойност – **boolean**. Ако въведеното число удовлетворява условието, което описахме по-горе, да е между 0 и 59 включително, методът ще върне като резултат **true**, иначе – **false**:

```
publicstaticboolean validateMinutes(int minutes) {
```

```
        boolean result = (minutes >= 0) && (minutes <= 59);
        return result;
}
```

След като сме готови с най-сложната част от задачата, декларираме метода **main()**. В тялото му, извеждаме въпроса, който беше указан в условието на задачата – "**What time is it?**". След това с помощта на класа **Scanner**, вземаме от потребителя числата за часове и минути, като резултатите ги съхраняваме в целочислените променливи, съответно **hours** и **minutes**:

```
Scanner input = new Scanner(System.in);
System.out.println("What tidme is it?");
System.out.print("Hours: ");
int hours = input.nextInt();
System.out.print("Minutes: ");
int minutes = input.nextInt();
```

Съответно, резултата от валидацията го съхраняваме в променлива от тип **boolean** – **isValidTime**, като последователно извикваме методите, които вече декларирахме - **validateHours()** и **validateMinutes()**, като съответно им подаваме като аргументи променливите **hours** и **minutes**. За да ги валидираме едновременно, обединяваме резултатите от извикването на методите с оператора за логическо "и" – **&&**:

```
boolean isValidTime =
        validateHours(hours) &&validateMinutes(minutes);
```

След като сме съхранили резултата, дали въведеното време е валидно или не, в променливата **isValidTime**, го използваме в условната конструкция **if**, за да изпълним и последния подпроблем от цялостната задача – извеждането на информация към потребителя дали времето, въведено от него е валидно или не. С помощта на **System.out**, ако **isValidTime** е true, в конзолата извеждаме "**The**

time is HH:mm now.", където **HH** е съответно стойността на променливата **hours**, а **mm** – тази на променливата **minutes**. Съответно в **else** частта от условната конструкция извеждаме, че въведеното време е невалидно – "**Incorrect time!**".

Ето как изглежда изходът от програмата при въвеждане на коректни данни:

```
What time is it?
Hours: 17
Minutes: 33
The time is 17:33 now.
```

Ето какво се случва при въвеждане на некоректни данни:

```
What time is it?
Hours: 33
Minutes: -2
Incorrect time!
```

Утвърдени практики при работа с методи

- Всеки метод трябва да решава самостоятелна, добре дефинирана задача. Това свойство се нарича **strong cohesion**. Фокусирането върху една, единствена задача позволява кодът да бъде по-лесен за разбиране и да се поддържа по-лесно. Един метод не трябва да решава няколко задачи едновременно!
- Един метод трябва да име, което описва какво прави той. Примерно метод, който сортира числа, трябва да се казва **sortNumbers()**, а не **number()** или **processing()** или **method2()**. Ако не можете да измислите подходящо име за даден метод, то най-вероятно методът решава повече от една задачи и трябва да се раздели на няколко отделни метода.

- Имената на методите е препоръчително да бъдат съставени от глагол или от глагол и съществително име (евентуално с прилагателно, което пояснява съществителното), примерно **findSmallestElement()** или **sort(int[] arr)** или **readInputData()**.
- Имената на методите в Java е прието да започват с малка буква. Използва се правилото **camelCase**, т.е. всяка нова дума, която се долепя в задната част на името на метода, започва с главна буква.
- Един метод или трябва да свърши работата, която е описана от името му, или трябва да съобщи за грешка. Не е коректно методите да връщат грешен или странен резултат при некоректни входни данни. Методът или решава задачата, за която е предназначен, или връща грешка. Всякакво друго поведение е грешно.
- Един метод трябва да бъде минимално обвързан с обкръжаващата го среда (най-вече с класа, в който е дефиниран). Това означава, че методът трябва да обработва данни, идващи като параметри, а не данни, достъпни по друг начин и не трябва да има странични ефекти (например да промени някоя глобално достъпна променлива). Това свойство на методите се нарича **loose coupling**.
- Трябва да се избягват методи, които са по-дълги от "един екран". За да се постигне това, логиката имплементирана в метода, се разделя по функционалност на няколко по-малки метода и след това тези методи се извикват в "дългия" до момента метод.
- Понякога, за да се подобри четимостта и прегледността на кода, е добре функционалност, която е добре обособена логически, да се отделя в метод. Например, ако имаме метод за намиране на лице на квадрат, процесът на пресмятане на квадрат на едно число може да се дефинира в отделен метод и след това, този нов метод, да се извика от метода, който пресмята лицето на фигурата квадрат. Разбира се, това ще ни даде възможност да

преизползваме метода за намиране на квадрата на едно число и на други места, когато ни е нужно.

Упражнения

1. Напишете метод, който при подадено име отпечатва в конзолата "**Hello, <name>!**" (например "**Hello, Peter!**"). Напишете програма, която тества този метод.
2. Създайте метод **getMax()** с два целочислени (**int**) параметъра, който връща по-голямото от двете числа. Напишете програма, която прочита три цели числа от конзолата и отпечатва най-голямото от тях, използвайки метода **getMax()**.
3. Напишете метод, който връща английското наименование на последната цифра от дадено число. Примери: за числото **512** отпечатва "**two**"; за числото **1024** – "**four**".
4. Напишете метод, който намира колко пъти дадено число се среща в даден масив. Напишете програма, която проверява дали този метод работи правилно.
5. Напишете метод, който проверява дали елемент, намиращ се на дадена позиция от масив, е по-голям, или съответно по-малък от двата му съседа.

12. Заключение

Считаме, че поставените цели са изпълнени. В настоящия си вид дипломната работа представлява съвременно ръководство за обучение по събитийно програмиране на **Java.** Личният принос на автора се състои в адаптацията на учебното съдържание към средата на **Java** и в съставянето на конкретни задачи и упражнения след срочните единици.