Въведение в Java

1. Обектно-ориентирано програмиране.

Езиците за ООП скъсяват дистанцията между създаването на отделните елементи на една сложна система. В тях основна програмна единица е класът и неговият конкретен представител – обектът. Класът обединява в едно данни и методи, които работят с тези данни.

ООП – езиците поддържат 4 основни механизма, които ги отличават от процедурните езици:

- по-високо ниво на абстракция на данните, т.е. нови структури от данни. Програмистът може да определя собствени типове данни, които се моделират посредством класове.
- механизъм за изграждане на йерархични структури от данни. Чрез наследяването на един клас от друг производният клас наследява модела данни на класа, който наследява и в същото време може да добави нови членове към този модел
- механизъм за управление на по-високо ниво на структурите от данни
- механизъм за дефиниране на операции от програмиста

2. Характеристики на Java.

Java е обектно-ориентиран, преносим, сигурен и независим от платформата език за програмиране.

Основните му характеристики са:

- Програмите, написани на Java могат да бъдат изпълнявани на всякакви платформи (операционна система + хардуер, за които има създадени виртуални машини) Затова говорим за неговата преносимост. Java спазва идеологията Write One/Run Anywhere. Как се постига това? – След създаването си, програмата се компилира до байткод. За всяка отделна платформа е създадена конкретна виртуална машина на Java, интерпретатор, който изпълнява генерирания байткод.
- Програмите на Java са малки по обем, защото носят информация, предназначена само за ВМ. Конкретната ВМ, от своя страна "знае" как да преобразува програмите до машинни команди за съответната платформа на която се стартират те.
- Java е бавна, защото са необходими време и системни ресурси програмата на Java да бъде преобразувана от ВМ до машинни команди за платформата. Но за конкретните платформи са създадени JIT-компилатори, които могат да преобразуват Java-код до изпълнима програма. JIT- компилатора се стартира на машината на крайния потребител точно преди интерпретатора на Java. Ако види код, той го подава на интерпретатора, но освен това го компилира и съхранява

получения прост код. Ако програмата се върне към тази част и компилаторът види същия код отново, той не се обръща към интерпретатора, а стартира простия код.

- Създателите на Java (Sun Microsystems) предоставят безплатно основния софтуер за Java JDK заедно с пълна документация на класовете и действията им, които могат да бъдат свалени от сайта на компанията – http://java.sun.com. Java спазва идеологията на отворения код (Open Source).
- Java предоставя множество от стандартни класове (модули) – Java Development Kit, които могат да бъдат използвани в програмите. Поради предното свойство, от различни организации, се създават все повече класове, които стават стандартни (като се включат в JDK), което води до бързо увеличаване на възможностите на езика за създаване на всякакъв вид приложения, използващи последните новости в най-различни области.
- Java е ОО език за програмиране.

3. Компилатор на Java и Виртуална машина на Java

Компилаторът на Java е програма – javac.exe, която се намира в bin директорията на JDK. Преобразува програмата в т.нар. byte code, който се явява входен език за ВМ на Java.

Пример:

- javac HelloWorld.java – записва се името на файла и разширението задължително
- javac *.java – компилират се всички файлове в текущата директория

Резултатът от компилирането е файл със същото име и разширение class - HelloWorld.class.

ВМ на Java изпълнява компилираните програми. ВМ на Java – java.exe, също се намира в bin директорията на JDK.

Тя търси метода “public static void main(String args[])“, от който започва изпълнението на програмата.

Пример:

- java HelloWorld – изпълнява се class-файла, но той се подава като параметър на ВМ без да се указва разширението му.

4. Създаване и стартиране на програма

Средства – Notepad или Wordpad, DOS Prompt.

За да се стартира програма, трябва да са определени някои системни променливи

- в PATH трябва да е зададен път до bin директорията;

- трябва да се зададе променливата CLASSPATH, в която се указват пътищата до класовете, които искаме да използваме, в частност до стандартните класове на Java, намиращи се архивния файл - C:\jdk1.3.0_02\jre\lib\rt.jar и до текущата директория.

Пример: CLASSPATH = .;C:\jdk1.3.0_02\jre\lib\rt.jar.

Отделните пътища се разделят със символа “;”

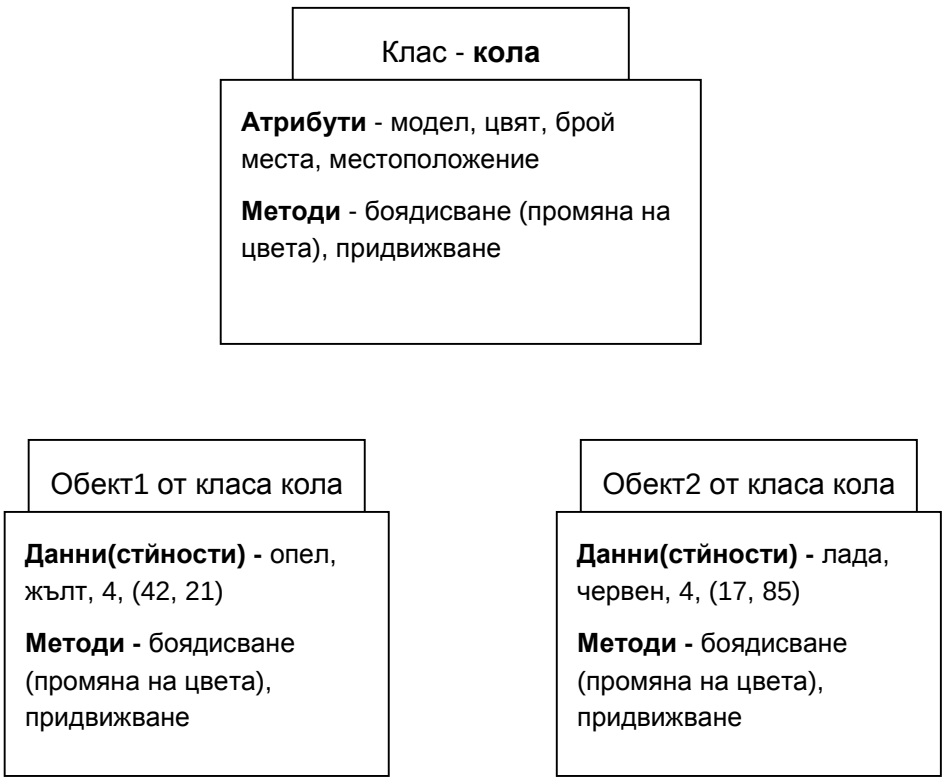
5. Езикът Java

В програмния език Java програмите са изградени от класове. Класовете моделират понятия от реалния свят, представляват един абстрактен модел от данни, докато обект на даден клас е конкретно представяне на този абстрактен модел. От една дефиниция на клас можете да създадете произволен брой обекти, известни като екземпляри (инстанции) на този клас. Класът съдържа членове, основните видове на които са полета и методи. Полетата са променливи с данни, принадлежащи или на самия клас или на обекти от този клас; те определят състоянието на обекта или класа. Методите са колекция от оператори (изпълними редове), които извършват операции върху полетата за да променят състоянието. Операторите дефинират поведението на класовете: те могат да присвоят стойности на полета или други променливи, да изчислят аритметични изрази, да извикат методи и да управляват реда на изпълняване.

Според ОО-концепция един клас може да наследява клас (наречен родителски) или да използва обекти (които са инстанции на друг клас).

Синоними – атрибути, характеристики, свойства, променливи на класа

Синоними – действия, операции, функции, процедури, методи



Фиг.1. Класове и обекти

Типове класове:

- **Приложения** (програми, applications)
    - с графичен интерфейс;
    - без графичен интерфейс;

Стартират се в команден интерпретатор (на операционната система или на някаква среда – Jbuilder, Kawa). Приложенията могат да се изпълняват само на локалния компютър (на който се намира техния код) и нямат ограничения за достъп до твърдия диск.

Приложенията без графичен интерфейс могат да извеждат информация само в изхода на командния интерпретатор (Например DOS Prompt), а тези с графичен интерфейс са обикновени GUI приложения. Разликата при създаването на двата вида е в различните стандартни класове, които се използват.

- **Аплети** (applet) – това са малки приложения, които се изпълняват обикновено в браузерите (или чрез помощната програма appletviewer). Имат графичен интерфейс. Не позволяват произволен достъп до паметта (оперативната памет и твърдия диск) на компютъра, на който се стартират.

- **Обикновени класове** – инстанции от тези класове могат да се използват от други класове.

Засега ще разглеждаме само програми без графичен интерфейс.

Пример Hello World:

```
package javaforbeginners;

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

Всеки клас се записва във файл със същото име и разширение .java.

Чрез предпочитания си текстов редактор запишете този програмен сорс код във файл със име HelloWorld.java. След това стартирайте компилатора за да компилира сорса на тази програма в байткод - „машинният език" за виртуалната машина на Java. Резултатът е файл с разширение .class.

javac HelloWorld.java

За да стартирате програмата трябва да да напишете командата с параметър името на .class-файла, като разширението се пропуска.

java HelloWorld

Това изпълнява метода main на HelloWorld. Щом стартираме програмата, тя извежда

Hello, world!

Сега вече имате малка програма, която прави нещо, но какво всъщност прави?

Програмата декларира един клас, наречен HelloWorld, с един единствен член: метод, наречен main. Членовете на класа се намират между фигурните скоби { и }, следващи името на класа. Методът main е особен метод: методът main на клас, ако е деклариран

както е показано, се изпълнява, щом стартирате класа като приложение. При изпълняването му, методът main може да създаде обекти, да изчисли изрази, да извика други методи и да направи всичко друго, необходимо за дефиниране на поведението на приложението.

Методът main е деклариран като public - така, че всеки да може да го извика (в случая - Виртуалната машина на Java) - и static, което означава, че методът принадлежи на класа, а не е асоцииран с конкретен екземпляр на класа.

Непосредствено преди името на метода е указан връщания тип. Методът main е деклариран като void защото той не връща стойност и поради това няма връщан тип.

След името на метода има списък от параметри на метода – поредица от нула или повече чифтове от тип и име, разделени чрез запетаи и затворени в кръгли скоби (и). Единственият параметър на метода main е един масив от обекти String (низове), наречен с името args. Означаването на масивите става чрез квадратни скоби [] след името на типа. В този случай args ще съдържа аргументите на програмата от командния

ред, с който тя е била извикана. За масивите и низовете ще говорим по-нататък. Името на метода, заедно със списъка на неговите параметри, връщаната стойност, възможните модификатори (например public и static) и списъка на изхвърляните изключения образуват декларацията на метод.

В този пример тялото на main се състои от единствен оператор, който извиква метода println - точката и запетая слагат край на оператора. Метод се вика, като се запише име на обект (в случая System.out - полето out на класа System) и името на метода (println), разделени от точка (.).

HelloWorld използва метода println на обекта out за да отпечата низ (терминиран с нов ред) в потока на стандартния изход. Отпечатва се низовият литерал „Hello World", който беше предаден като аргумент на println. Низовият литерал представлява редица от символи, заградени в двойни кавички „ и „.

6. Идентификатори

Идентификатор – последователност от символи, избрани от програмиста, с която се представят променливи, константи, класове, обекти, етикети или методи. След като веднъж един идентификатор е създаден, той представя един и същ обект във всяка част на блока, в който е обявен.

Правила при създаване на идентификатор:

- Първият символ е буква, а следващите могат да бъдат букви и цифри
- Символът _ и символът $ са приети за букви и могат да се използват в идентификаторите, включително като първи символ
- Прави се разлика между малки и големи букви
- Добър стил е идентификаторите да са описателни

- Добър стил е идентификаторите на класовете да започват с главна буква, а тези на атрибути и методи с малка.
- Добър стил при съставни идентификатори е началото на всяка съставна дума да е с главна буква

Пример:

Правилни идентификатори: HelloWorld, counter, HotJava$, first_point

Неправилни идентификатори: 9hello, count&add, Hot Java, 2010, one-two

7. Коментари

Има три типа коментари:

Текстът, намиращ се между /* и */, бива игнориран от компилатора. Този вид коментар може да заеме част от реда, цял ред или най-често, да бъде многоредов коментар. За коментари, заемащи единичен ред или част от ред, можете да използвате и означението //, което указва на компилатора да игнорира всичко от тук до края на реда.

Третият вид коментар се разполага между /** и */, нарича се документиращ коментар и описва следващите го декларации. Такъв коментар може да бъде извлечен чрез различни инструменти, които го използват за автоматично генериране на документация за класа. Например с javadoc.exe от него може автоматично да се създаде HTML-документация.

8. Пакети

Класовете могат да бъдат групирани в пакети. Понятието пакет съответства на директория, така както клас съответства на файл. Всички класове, принадлежащи на един пакет, се разполагат в една директория. Групирането се прави с цел осигуряване уникалност на имената на класовете, т.е. възможно е съществуването на два класа с еднакви имена, но в различни пакети.

В JDK, разпространен от Sun Microsystems, се съдържат множество от стандартни пакети. Един пакет може да има и подпакети. В тях са групирани класове осигуряващи разнообразни възможности: за работа с аплети, форми, дата, низове, числа, масиви, нишки, бази данни…

За да могат да се използват тези класове в едно приложение, те трябва да бъдат декларирани (импортирани) в дефиницията на класа му (на приложението).

Импортиране на един клас:

```
import java.util.Hashtable - (Класа Hashtable от пакета java.util).
```

Импортиране на всички класове от един пакет:

```
import java.util.*.
```

За да се декларира, че един клас е от даден пакет, трябва да се използва ключовата дума **package**:

```
package javaforbeginners;
```

Ако декларациите package и import съществуват, то те трябва да са преди декларацията на класа, в указаната последователност. Например:

```
package javaforbeginners;
import java.util.*;
public class Example
{
   …….тяло на класа….
}
```

Класовете на основния пакет java.lang не се нуждаят от импортиране - java.lang.String, java.lang.Boolean.

9. Променливи и видимост

Променливи могат да се задават на всяко място в клас на Java. Те са видими в блока, в който са декларирани (пример).

- променливи на класовете (атрибути) – обикновено се декларират преди методите. Видими са във всички методи на класа. Атрибутите са променливи с данни, асоциирани с класа и неговите обекти, те представляват състоянието на класа.
- променливи на методите - видими са и могат да се използват само в методите, в които са декларирани.
    - параметри
    - обикновени

Пример – четене на символ от клавиатурата:

```
package javaforbeginners;
import java.io.*;

public class ReadHello
{
  public static void main(String args[])
  {
    int inChar;
    System.out.print("Enter a character:");
    try
    {
      inChar = System.in.read();
```

```
      System.out.println("You entered " + (char)inChar);
    }
    catch (IOException e)
    {
      System.out.println("Error reading from keyboard!");
    }
  }
}
```

Пример – извеждане на текст в конзолата:

```
package javaforbeginners;
public class HelloFriend
{
  static String friendName = "John";
  public static void main(String[] args)
  {
    System.out.println("Hello, " + friendName);
  }
}
```

10. Методи

Методите представляват мини-програми и извършват част от работата на цялата програма. Съдържат изпълнимия код на класа. Те се състоят от декларация и тяло (тялото е заградено във фигурни скоби).

Методите може да връщат стойност – обект от всеки възможен клас или прост тип (за цели числа – byte, short, int, long; реални числа - float , double; знак- char; boolean, със стойности true и false).

Тип на връщаната стойност void, означава, че метода не връща стойност. Връщане на стойност става с ключовата дума return, последвана от стойността, която желаем да върнем.

Пример – статичен метод, връщащ стойност:

```
package javaforbeginners;
public class IncExample
{
  public static int inc(int number)
  {
      return number + 1;
  }
  public static int add(int number, int anotherNumber)
  {
      return number + anotherNumber;
  }
}
```

Пример - извикване на статичен метод, връщащ стойност.

```
package javaforbeginners;
public class IncExampleProgram
{
  public static void main(String[] args)
  {
    int temp = IncExample.add(6,2);
    System.out.println("6+2 = " + temp);
    System.out.println("6++ = " + IncExample.inc(6));
  }
}
```

Конструкторът служи за създаване на инстанция на клас (обект). В него обикновено се инициализират атрибутите на класа.

Обект се създава чрез конструкцията

Име_на_клас име_на_обект = *new* конструктор_на_класа_с_фактически_параметри;

Към public променливи и методи може да се обръщаме така, но при други спецификатори за достъп такива обръщения може да са невалидни.

Синтаксис

Синтаксисът на Java (и на всеки формален език) се състои от:

- Азбука –Java използва 16 битова кодова таблица – Unicode – в която първите 256 байта са символите на ASCII таблицата. Изобразяването на повечето от тези символи става със специални escape последователности: \uh, където h е шестнадесетично число (с брой символи до 4);
- Лексика – правила за съставяне на думи/лексеми от азбуката; Java прави разлика между малки и големи символи (Например, думите class и Class са различни). Типове думи на езика са:
    - Ключови (запазени) думи – class, public, package, import, static, this, super…
    - Числени и символни константи – 12, 12.2, “низ”;
    - Идентификаторите са имена, чрез които се обозначават класове, обекти, методи, променливи.
    - Непозволени символи в имената са символи за аритметични и булеви операции, например “-”, “+”, “:”, “&”, “<”…
    - Разделителите са символите, с които се разделят лексемите една от друга. Те са интервал, табулация, край на ред.
- Граматика – правила за съставяне на изречения от думите на езика. Всяко изречение завършва със символа “;”.

1. Типове данни

Типовете данни в Java са две категории: примитивни и съставни. Съставните типове данни обхващат масивите, класовете и интерфейсите.

1.1. Примитивни типове данни

В Java има осем типа примитивни типа данни, които са показани в следващата таблица заедно с някои техни характеристики.

Тип	**Описание**
boolean	Стойности true и false
byte	8-битово цяло число. Интервал: -128 до 127
short	16-битово цяло число. Интервал: -32768 до 32767

char	16-битов Unicode знак или цяло число. Интервал: 0 до 65535
int	32-битово цяло число. Интервал: -2147483648 до 2147483647
long	64-битово цяло число. Интервал: -2^{63} до $2^{63}-1$
float	32-битово число
double	64-битово число

За всеки примитивен тип съществува съответен клас, чието име е същото като на примитивния тип, но започва с главна буква: Boolean, Byte, Short…

1.1.1. Променливи от тип boolean

Това е най-простият тип данни, осигурен от Java. Променливите от този тип имат само 2 възможни стойности: true или false. Променливите от тип boolean се използват най-често за запазване на информация за състоянието на някой обект. Например лист хартия може да бъде на масата, но може и да не бъде на масата.

1.1.2. Целочислени типове

Това са byte, short, int, long, char (char може да се третира и като символ). Представят числа в различен интервал от стойности. Има различни причини за ползване на всеки от типовете, предимно от гледна точка да не се заема ненужно памет. Например при представяне на проценти в програмата е ненужно избиране на тип, който далеч надхвърля границите на нужния ни интервал 0-100.

1.1.3. Променливи от тип char

Символите в Java са специален тип. Той може да бъде третиран като 16-битово цяло число без знак (виж целочислени типове) или като символ от кодовата таблица Unicode.

1.1.4. Типове float и double

Тези типове се използват за представяне на числа, които съдържат десетична точка. Разбира се всички цели числа също могат да се представят така, например 5 като 5.0. Числата с плаваща запетая имат три особени стойности:

1.1/0 // Изход: Infinity – плюс безкрайност

-1.1/0 // Изход: -Infinity – минус безкрайност

0.0/0 // Изход: NaN – невалидно число

1.1.5. Операции с примитивни типове данни

a) Деклариране на променливи от даден тип (и присвояване на стойност):

Име_на_тип име_на _променлива [= стойност] {, име_на _променлива [= стойност]};

Пример:

```
double d1, d2 = 3, d3 = 3.3;
char c1 = 'a', c2 = 56;
```

b) Присвояване на стойност на променлива

Става с оператора =.

Пример:

```
boolean  onTheTable;
int i;
double d;
char c;
onTheTable = true;
i = 678;
d = 97593725.25364;
c = 'a';
```

c) Аритметични оператори +, -, *, /, %

Операндите са числа. Резултатът е число.

% е оператор за остатък при делене на две целочислени числа.

Примери:

```
10%3 = 1
10%4=2
int j = 12;
int k = 3;
int result = j + k;
```

При делене на две цели числа се получава цяло число.

Пример:

```
int i1 =10, i2=3;
System.out.println(i1/i2);  // Изход: 3
```

d) Други оператори за присвояване +=, -=, *=, /=, %=

(+= присвояване и събиране, -= присвояване и изваждане, *= присвояване и умножение, /= присвояване и деление, %= присвояване и остатък от деление)

Операндите са числа. Резултатът е число.

Пример:

```
byte j = 60;
short k = 24;
int l = 30;
int m = 12;
int result = 0;
result += j;  // на result се присвоява 60 ( 0 + 60)
result += k; // на result се присвоява 84 ( 60 + 24)
result /= m; // на result се присвоява 7 ( 84 : 12)
result -= l; // на result се присвоява -23 ( 7 - 30)
result = -result; // на result се присвоява 23
result %= 2; // на result  се присвоява 1 (остатъка от 23 : 2)
result *= 3; // на result  се присвоява 3 ( 1 * 3)
```

e) Други оператори за числа ++, --

(++ инкрементиращ оператор, -- декрементиращ оператор)

Операндите са числа. Резултатът е число.

Пример:

```
Ако имаме double d =1.1;
d++; // d получава стойност 2.1
d--; // d получава стойност 0.1
```

f) Оператори за сравнение ==, !=, <, <=, >, >=

Операндите са числа. Резултатът е логическа стойност.

Пример:

```
5==3 // false
```

g) Логически оператори && (логическо и), || (логическо или), ! (логическо не)

Операндите са логически стойности. Резултатът е логическа стойност.

A	B	A&&B	A||B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

1.2. Константи

Константите са стойности, които не се изчисляват и остават непроменени до края на програмата. Именувана константа е константна стойност, която назоваваме по име. Именувани константи се дефинират, като се декларират полета от подходящ тип и се инициализират с подходяща стойност. Това само по себе си не дефинира константа, а поле, чиято стойност може да бъде променяна чрез оператор за присвояване. За да направим стойността константна, декларираме полето като final (окончателно). Поле или променлива, декларирани като final не могат да променят стойността си след инициализация – те са немутиращи. Освен това понеже не искаме именуваните полета да бъдат асоциирани с екземпляри на класа, то ги декларираме като static.

Пример:

```
Class Suit
{
  final static int CLUBS = 1;
  final static int DIAMONDS = 2;
  final static int HEARTS = 3;
  final static int SPADES = 4;
}
```

За да използваме статичен член на клас, пишем името на класа, последвано от точка и името на члена – Suit.HEARTS, Suit.SPADES и т.н.

```
package javaforbeginners;
public class HelloWorldConstant
{
  final static String TEXT = "Hello World!";
  public static void main(String[] args)
  {
    System.out.println(HelloWorldConstant.TEXT);
  }
}
```

1.3. Масиви

Масив е наредена последователност от еднотипни елементи. Той е лесен начин да се обединят няколко променливи в една, особено при данни, които могат да се индексират.

Съществуват три основни действия с масивите:

1) Деклариране на масив (два начина):
    - тип_на_елементите_на_масива [] име_на_масив;
    - тип_на_елементите_на_масива име_на_масив[];

Пример:

```
double MyDoubleArray[]; // double [] MyDoubleArray;
```

2) Заделяне на памет и определяне на размера:

   име_на_масив = **new** тип_на_елементите_на_масива[брой_елементи]

   За горния пример:

   ```
   MyDoubleArray = new double[10];
   ```

3) Достъп до елементите на масива:

   име_на_масив[индекс_на_елемент]

Първият елемент на масива винаги е с номер 0. Последният елемент е с индекс (броят на елементите на масива-1).

За горния пример:

```
MyDoubleArray[0]=3; //  Запис на стойност в 1-я елемент.
MyDoubleArray[4]=3; //  Запис на стойност в 5-я елемент.
MyDoubleArray[9]=3; //  Запис на стойност в последния (10-я) елемент.
int i1 = MyDoubleArray[4]; // Извличане на стойност от елемент на масива.
```

За масиви с елементи от прост тип може да се задава стойност на масива при декларирането му (без да е нужно специално заделяне на памет и определяне на размера).

Пример:

```
int iArray [] = {1, 5, 6}; // Автоматично се заделя памет за масив с 3
елемента.
```

Индексът трябва да е от тип int или да се преобразува до този тип, като резултат от това най-големият размер на масив в Java е 2147483647, това е максимумът, дефиниран от стандарта.

1.4. Двумерни масиви.

Пример:

```
int Matrix1[][] = new int[3][3];
int Matrix2[][] = {{11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

1.5. Низове.

Класът String е клас за работа с низове. В Java са дадени улеснения за работа с низове – не е нужно явно да се извиква конструктор на класа, за да бъде създадена инстанция на класа – създаването и инициализирането на низов обект може да стане с един оператор. При създаването на обект не е необходимо да се задава дължината му.

```
String s = new String();
String s = “”;
String s = new String(“This is a string”);
String s = “This is a string”;
char charArray[] = {‘a’, ‘b’};
String s = new String(charArray);
```

Низове може да се конкатенират с променливи от всички примитивни и съставни типове, чрез оператора +. При това за съставните типове компилатора на Java извиква метода toString(), който е дефиниран в класа Object. Този метод може да бъде предефиниран във всеки нов клас.

Пример:

```
package javaforbeginners;

public class StringDemo
{
  public static void main(String[] args)
  {
    String myName = "John";
    String occupation = "Computer specialist";
    myName = myName + " Atanasov";
    System.out.println("Name = " + myName + " (" + occupation + ")" );
    System.out.println(myName.length());
    System.out.println(myName.charAt(2));
  }
}
```

1.5.1. Сравняване на низове:

Символните низове в Java са обекти, за сравняване на които можем да ползваме == или метода equals. Например:

```
String a = new String(“Foo”);
String b = new String(“Foo”);
```

Резултатът от сравнението a==b ще е false, защото това са различни обекти, независимо че стойностите им са равни. Сравнението a.equals(b) ще върне true, защото стойностите им са равни. Класът String осигурява метод за сравнение, който не прави разлика между малки и големи букви:

```
public Boolean equalsIgnoreCase(String anotherString)
```

2. Изрази

Изразът е съвкупност от оператори и операнди, подчинени на синтактични правила. Изразите се използват, за да осъществяват операции и манипулации върху променливи или стойности.

Пример:

```
1 + a * 3 – ( b+c)
```

Редът на изчисляване на израза зависи от приоритетите на операциите. Приоритетът може да се променя с използването на скоби.

За осъществяване на операции между разнотипни операнди и за определяне на типа на израза се налага да се извърши преобразуване на типовете. В Java съществува два типа преобразуване на типове: явно и неявно.

Когато се осъществяват операции между цели числа Java, автоматично преобразува “по-малкия” тип до по-големия (или до още по-голям). При работа с реални числа ако поне един операнд е от тип double всички операнди се преобразуват до double иначе до float.

Понякога е нужно да сме сигурни, че даден операнд е от точно определен тип. Тогава се използва явно преобразуване на типове. Съществува оператор за преобразуване на тип: *(име_на_тип)*.

Пример:

```
float x = 2.0;
float y = 1.7;
x = ((int) (x/y)*y);
```

3. Методи

Методите изпълняват същите задачи, каквито и функциите в Pascal, C++.

Декларация на метод:

[спецификатор_за_достъп] [модификатор] [тип_на_връщана_стойност] име_на_метод ([списък_с_параметри]) [throws списък_с_изключения];

3.1. Спецификатор за достъп

Той се използва за ограничаване на достъпа до метода от методи на други класове (Без значение от спецификатора, методът е достъпен от всички методи от класа, в който е създаден).

Спецификатор	**Описание**
public	Методът е достъпен от всички класове, без значение от произхода им и пакета, в който се намират.
protected	Методът е достъпен само за текущия пакет и наследниците на класа.
friendly	Методът е достъпен за всички класове от текущия пакет. Това е стойността по подразбиране. Извън текущия пакет се счита private.
private	Методът е достъпен единствено за методите на класа.

Достъп от текущия пакет до членовете на класа:

Спецификатор	**Наследяване**	**Достъп**
public	да	да
protected	да	да
friendly (default)	да	да
private	не	не

Достъп от друг пакет до членовете на класа:

Спецификатор	**Наследяване**	**Достъп**
public	да	да
protected	да	не
friendly (default)	не	не

private	не	не

3.2. Модификатор на метод

Той установява начина на взаимодействие на други методи с този метод.

1) static

Пример:

```
public class StaticExample
{

    static double staticMethod()
   {
      return 2.1;
    }
}

public class StaticTest
{
  public static void main(String args[])
  {
    double d1, d2;
    d1 = StaticExample.staticMethod();
    StaticExample sE = new StaticExample();
    d2 = sE.staticMethod();
    System.out.println(d1 + " " + d2);
  }
}
```

Променливите d1 и d2 имат еднакви стойности. Двата вида обръщения към статични методи на класа водят до един и същи резултат.

2) abstract

Абстрактните методи само се декларират в текущия клас. Дефинициите трябва да се намират в наследниците на класа. Класът, в който се намира абстрактен метод трябва също да бъде деклариран като абстрактен.

Пример:

```
public abstract class Car
{
……
    abstract double maxSpeed();
……
}

public class Ford extends Car
{
```

```
    public double maxSpeed()
    {
        return 230;
    }
}

public class Lada extends Car
{
   public double maxSpeed()
   {
     return 160;
   }
}
```

3) final

Този модификатор забранява на всеки наследник на класа да припокрие метода. Тази възможност увеличава степента на изолация между класовете и може да се даде функционалност на един клас, която да не може да се промени в наследниците му.

4) native

Този модификатор информира компилатора, че метода е написан на друг език за програмиране.

5) synchronized

Този модификатор спомага за запазване верността на данните при едновременен достъп до едни и същи данни. Той се използва при работа с паралелни разклонения.

Един метод може да има няколко модификатора, но не и всички наведнъж. Например абстрактен метод не може да е final, защото няма да може да бъде дефиниран в наследника на класа.

3.3. Тип на връщана стойност

Като тип на връщана стойност на метод може да бъде записан всеки тип. Когато метода не връща стойност се записва ключовата дума void. За конструктор не се записва тип на връщана стойност. Връщане на стойност става с ключовата дума return.

3.4. Списък от параметри

Указва каква последователност от информация ще се подава на метода при неговото извикване.

списък_с_параметри = [име_на_тип1 име_на_формален_параметър1 [,име_на_тип2 име_на_формален_параметър2,…]]

3.5. Предаване на параметри

Предаването на примитивен тип е винаги по стойност. Стойността на фактическия параметър не се променя. Съответните класове на примитивните типове както и обекти от класа String също не променят стойността си.

Предаването на обект е винаги по име (адрес, псевдоним). Стойността на фактическия параметър се променя.

Пример:

```
public class ParametersExample
{
  long l;
}

public class TestParameters
{
  public static void changeValues(long l, ParametersExample pe)
  {
    l++;
    pe.l++;
  }

  public static void main(String args[])
  {
    long l = 5;
    ParametersExample pe = new ParametersExample ();
    pe.l =5;
    System.out.println( "Before: l= " + l + "; pe.l= " + pe.l);
    changeValues(l, pe);
    System.out.println("After: l= " + l + "; pe.l = " + pe.l);
  }
}
```

4. Блокове от оператори

Всеки блок от оператори се загражда във фигурни скоби.

```
{
   op1;
   op2;
   ……..
   opn
}
```

Един оператор може да представлява блок от оператори или да включва в тялото си блок от оператори. Един блок представлява един оператор.

Всяка променлива е видима (достъпна) само в блока, в който е дефинирана.

Управление изпълнението на програма

1. Конструкция if

Синтаксис:

```
if(логически_израз)
    оператор;
```

Проверява се стойността на логическия израз. Ако тя е true, то последващият блок от оператори се изпълнява, в противен случай операторът след if се пропуска.

Пример:

```
String s = "TesT"
if(s.equalsIgnoreCase("test"))
{
  System.out.println("s = test");
}
```

2. Конструкция if-else

Синтаксис:

```
if(логически_израз)
  оператор1;
else
  оператор2;
```

Проверява се стойността на логическия израз. Ако тя е true, то блокът от оператори след if се изпълнява, а този след else се пропуска, в противен случай операторът след if се пропуска, а се изпълнява операторът след else.

3. Оператор за цикъл while

Синтаксис:

```
while(логически_израз)
  оператор;
```

Проверява се стойността на логическия израз и докато тя е true операторът *оператор* се изпълнява.

4. Оператор за цикъл do-while

Синтаксис:

```
do
  оператор;
while(логически_израз);
```

Тук условието се проверява след изпълнението на *оператор*. Това гарантира, че този цикъл ще се изпълни поне веднъж, докато горния цикъл може да не се изпълни нито веднъж.

5. Оператор за цикъл for

Синтаксис:

```
for(инициализация; логически_израз; стъпка)
  оператор;
```

При изпълнение на цикъла първо се изпълнява инициализацията, еднократно. В началото на всеки цикъл, преди изпълнението на *оператор* се изчислява логическия израз. Операторът се изпълнява само ако стойността на израза е true. Иначе изпълнението на програмата продължава след цикъла. След изпълнението на тялото на цикъла се изпълнява *стъпка. Инициализация* и *стъпка* може да са празни или да са съвкупност от оператори разделени помежду си със запетаи.

Пример:

```
for(int i=0; i<5; i++)
  System.out.println(i);
```

Обхождане на елементите на двумерен масив:

```
int i, j;
for(i = 0; i<3; i++)
{
  for(j = 0; j<3; j++)
    System.out.println(Matrix2[i][j] + ", ");
}
```

6. Конструкция switch

Синтаксис:

```
switch(израз)
{
  case value1: оператор1; [break;]
  case value2: оператор2; [break;]
  ……..
  [default : оператор0;]
}
```

Изразът може да е само от тип byte, short, char или int.

Стойността на израза се сравнява със стойностите след *case* и се изпълнява оператора след съвпадащата стойност. Ако не се използва break след изпълнението на оператора, се изпълнява следващия оператор в конструкцията и т.н. Ако не е намерено съответствие се изпълнява оператора след *default,* ако съществува.

7. Оператори за безусловен преход

- **break** – ако се използва в тялото на цикъл, без обръщение към етикет, то се прекъсва изпълнението на най-вътрешния цикъл;
- **continue** – този оператор се използва само в тялото на цикъл. Предизвиква прекъсване на изпълнението на текущата итерация и преминаване към следващата;
- **return** – прекъсва изпълнението на метода. Управлението на програмата се предава в точката на извикване на метода. Ако за метода не е дефиниран тип на връщана стойност оператора не се нуждае от операнд.

Задача: изчисляване на факториел.

```
public class Faktoriel
{
  public static void main(String[] args)
  {
    int n = 5;
    int faktoriel = 1;
    for (int i = 1; i <= n; i++)
      faktoriel *= i;
    System.out.println("n! = " + faktoriel);
  }
}
```

Задача: пример с масив

```
public class ArrayExample
{
  public static void main(String[] args)
  {
    String myArray[];
    myArray = new String[5];
    myArray[0] = "First";
    myArray[1] = "Second";
    myArray[2] = "Third";
    myArray[3] = "Fourth";
    myArray[4] = "Fifth";
    for (int i = 0; i < myArray.length; i++)
    {
      System.out.println(myArray[i]);
      if (myArray[i].equalsIgnoreCase("third"))
        break;
    }
  }
}
```

Задача: пример със switch

```
public class SwitchExample
{
  public static void main(String[] args)
  {
    int key = 1;
    switch (key)
    {
      case 1:
        System.out.println("One");
        break;
      case 2:
        System.out.println("Two");
        break;
      default:
        System.out.println("Default");
        break;
    }
  }
}
```

Задача: пример с do-while

```
public class DoWhileExample
{
  public static void main(String[] args)
  {
    int i = 0;

    do
    {
      System.out.println(i);
      i++;
    }
    while(i<5);
  }
}
```

Класове

Класовете са изграждащите блокове на всяка Java програма (и на всяка обектно-ориентирана структура). Класовете правят възможно създаването на обекти, които са в основата на обектно-ориентираното програмиране и дават възможност за капсуловане на данните, разделяйки информация и действия, специфични за обекта, от останалата част на кода. Те предоставят структурата на обектите и механизмите за произвеждане на обекти от дефинициите на класовете.

Обикновено класовете се разглеждат като начин за задаване на множество от данни и методи, които осъществяват операциите с тези данни, т.е класовете се състоят (дефинират) от променливи (атрибути) и методи. Атрибутите определят състоянието и дефиниционната област на класа, т.е. пространството на всички възможни стойности на екземплярите на класа – обектите. Методите са колекции от изпълним код, които са центъра на изчисленията и манипулират данните, съхранени в обектите, определят възможните операции с обектите, поведението на обектите.

1. Основни предимства на ООП

- Капсуловане (групиране) на данни и методи, които ги обработват. Това е възможност за ефективно изолиране и отделяне на информацията от останалата част на програмата. Със създаването на отделни модули, които веднъж вече са разработени като завършен клас, изпълняващ определени действия , можете да забравите сложността на реализацията и просто да използвате предоставените методи. Друга полза от капсулованието е, че с поставянето на данните в класове и създаването на методи за тяхното управление се осигурява програмата да не може да промени непозволено данните.
- След като веднъж сме създали даден клас може да го използваме лесно, без да се интересуваме от сложностите при създаването на отделните негови части.
- Наследяването предоставя, възможност за изграждане на нови програми като се използват възможностите на стари и се създадат само нови методи на класовете. Т.е. възможно е да се създава йерархия от класове, чрез която да се създават все по-сложни и по-сложни структури на базата на някоя проста структура.
- Полиморфизмът позволява наследници на даден клас да се разглеждат като инстанции на родителския си клас. Поведението на класовете може да е различно, но всички те притежават характеристиките и възможностите на родителя. Например: нека сте създали класа Vehicle и неговите наследници – класовете Truck и Bike. Въпреки, че камионите и велосипедите се различават много, те все пак са превозни средства. Следователно всичко, което можете да направите с инстанция на класа Vehicle, можете да направите и с инстанции на класовете Truck и Bike. В примера докато всеки камион и велосипед е превозно средство, превозното средство не е нито камион, нито велосипед. Поради тази

причина докато класовете Truck и Bike могат да се разглеждат като клас Vehicle, то не можете да извършите операция, специфична за класа Bike или класа Truck върху инстанция на Vehicle.

2. Декларация на клас

[модификатори] class име_на_клас [extends име_на_родителски_клас] [implements име_на_интерфейс {, име_на_интерфейс }]

3. Модификатори

Те определят какви са възможностите за използването на класа от други класове - те не оказват влияние при разработката на самия клас. Ако не бъде изписан модификатор се използва модификатора по подразбиране friendly.

Модификатор	**Описание**
public	Класове, декларирани като public са достъпни за всички обекти. Те могат да се използват от произволен обект и наследяват от произволен клас.
friendly	Модификатор по подразбиране. Определя, че класът може да се наследява и използва, но единствено обектите от същия пакет могат да използват методите и полетата на класа директно.
final	Класове, декларирани като final, не могат да бъдат наследявани.
abstract	Клас, който има недефинирани (абстрактни) методи, които задължително трябва да се дефинират в субклас. За такива класове не може да се създава инстанция с оператора new.

Един клас не може да бъде едновременно final и abstract.

4. Полета на класа (атрибути)

Променливите в класа се наричат полета или атрибути. Декларацията на атрибут се състои от име на типа, следвано от името на атрибута и незадължително инициализираща клауза, даваща начална стойност.

Модификатори на атрибутите на класовете - определят някои характеристики на атрибута.

Модификатор	**Описание**
static	Стойността на този атрибут е обща за всички инстанции на класа. Щом един атрибут се декларира като статичен, то съществува само едно негово копие, независимо колко екземпляра на класа сме създали. Статичните методи могат да използват само статични атрибути.
final	Указва, че стойността на атрибут не може да се променя след инициализирането и по време на изпълнение на програмата. Затова тя трябва да се зададе по време на деклариране на атрибута: final int i = 5;

```
public class StaticFieldExample
{
  public static int number = 6;

  public static void main(String[] args)
  {
     StaticFieldExample object1 = new StaticFieldExample();
     System.out.println(object1.number);
     object1.number = 7;
     StaticFieldExample object2 = new StaticFieldExample();
     System.out.println(object2.number);
     object2.number = 28;
     System.out.println(object1.number);

     if(object1 instanceof StaticFieldExample)
    {
       System.out.println("Yes");
    }
  }
}
```

Спецификатори за достъп до променливите на класовете - определят начина за достъп до атрибутите от други класове.

Спецификатор	**Описание**

friendly	Стойност по подразбиране. Атрибутите са достъпни за класовете от същия пакет.
public	Атрибутът е достъпен от всички класове.
protected	Атрибутът е достъпен от наследниците на класа.
private	Атрибутът не е достъпен извън класа си.

Константа се декларира с модификаторите static и final:

```
static final int i = 5;
```

5. Създаване на обекти

Когато простото инициализиране не е достатъчно, класовете могат да имат конструктори, които да инициализират полетата на обекта с някакви стойности. Конструкторите са блокове от оператори, които могат да инициализират обект преди да бъде върната референция към него (от new). Конструкторите имат еднакво име с името на класа, който инициализират. Подобно на методите те приемат нула или повече аргументи, но конструкторите на са методи и затова нямат връщана стойност. Конструктор може да извика друг конструктор от същия клас чрез извикването на this(списък с параметри) като свой **първи** оператор.

Списъкът с параметрите определя коя версия на конструктора ще бъде извикана в израза new.

6. Методи на класа

Съдържат кода, който разбира и променя състоянието на обекта. Разгледани в упражнение с тема “Синтаксис”

Извикване на метод: методите се извикват като операции върху обекти чрез референция и оператор точка (.).

```
референция.метод(аргументи);
```

Предефиниране на методи: Възможността два или повече метода да имат едно и също име, при условие че имат различен брой или тип на параметрите.

Статичните методи работят за целия клас, а не върху отделен екземпляр на класа. Могат да изпълняват задача обща за всички методи на класа. Те имат достъп само до статичните полета и методи на класа, защото обръщането към нестатични методи трябва да става чрез референция към обект.

7. Ключовата дума this

Тази ключова дума може да се използва за явно извикване на един конструктор в началото на друг. Специалната референция this може да се използва и в тялото на нестатичен метод, където тя сочи към текущия обект, за който е извикан методът. В статичните методи няма референция this, защото няма определен обект, върху който се оперира.

Референцията this се използва най-често, за да предадем референция към текущия обект като аргумент на други методи.

Пример:

```
Body (string BodyName, Body orbitsAround)
{
  this();
  name = bodyName;
  orbits = orbitsAround;
}
```

8. Оператор instanceof

Операторът instanceof определя дали левият операнд (обект) е инстанция на операнда отдясно (клас). Връща резултат логическа стойност.

Наследяване на класове

Една от главните ползи от ООП е възможността да наследяваме (разширяваме) съществуващ клас и по този начин да използваме неговите полета, методи, т.е кодът, написан за първоначалния клас да продължава да работи и в екземпляри и на подкласа. Когато разширяваме даден клас, за да създадем нов клас, новият разширен клас наследява **достъпните** полета и методи на суперкласа. По този начин се спестяват време, писане, търсене и поправяне на грешки. Ако не се промени нищо в новия клас, то той ще има същото поведение като родителя. По подразбиране всеки клас е наследник на класа java.lang.Object. Така че ако не сте указали наследяване, все пак класът наследява java.lang.Object.

Наследяването на клас може да се използва за различни цели. Най-често се използва за специализиране – тогава разширеният клас дефинира ново поведение и поради това става специализирана версия на своя суперклас.

Java не поддържа множествено наследяване.

Клас, който е наследяван – суперклас.

Клас, който наследява – субклас, разширен клас.

1. Полиморфизъм

Полиморфизъм е възможността обекти на даден клас да се представят като обекти на свой родителски клас (пряк родител или родител от йерархията на наследяванията за класа).

Това се осъществява чрез механизма за преобразуване на типове. Явно преобразуване на типове става чрез оператора typecast - (), където между двете скоби се записва име на тип.

Пример:

```
public class Car
{
  String name;

  public Car()
  {
    name = "";
  }
}

public class Ford extends Car
{
  String model;

  public Ford(String model)
  {
```

```
    name = "Ford";
    this.model = model;
  }
}

public class Lada extends Car
{
  public Lada()
  {
    name = "Lada";
  }
}

public class TestCar
{
   public static void main(String args[])
   {
     Ford ford = new Ford("Escort");
     Lada lada = new Lada();
     Car cars[] = {ford, lada};
     Lada anotherLada = new Lada();
     Car car = anotherLada;

     if (car instanceof Car)
     {
       System.out.println("Instance of Car");
     }

     if (car instanceof Lada)
     {
       System.out.println("Instance of Lada");
     }

     if (car instanceof Ford)
     {
       System.out.println("Instance of Lada");
     }

     System.out.println("List of cars:");
     for(int i = 0; i < cars.length; i++)
     {
       System.out.print((i+1) + ". " + cars[i].name);
       if(cars[i] instanceof Ford)
       {
         Ford temp = (Ford) cars[i];
         System.out.println(" - " + temp.model);
       }
       else
         System.out.println();
    }
  }
```

```
}
```

2. Конструктори в разширените класове

Обектът на разширения клас съдържа полета, които са наследени от суперкласа и полета, дефинирани локално в класа. За да се конструира обект от наследения клас трябва да се инициализират коректно и двете множества полета. Конструкторът на разширения клас може директно да извика един от конструкторите на суперкласа чрез ключовата дума super. Ако първият изпълним ред на конструктора на наследника не е извикване на конструктор на суперкласа или един от собствените конструктори на наследника, то автоматично се извиква безаргументният конструктор на суперкласа. Ако суперкласът не притежава безаргументен конструктор, то трябва да се извика друг явно, иначе компилаторът ще изведе съобщение за грешка.

Конструкторите не са методи и не се наследяват. Ако суперкласът има някакви конструктори и разширения клас трябва да има конструктори от същия вид, тогава разширеният клас трябва явно да декларира всеки конструктор, дори и ако всичко, което прави той, е да извика конструктор на суперкласа.

Пример:

```
public class SuperClass
{
  public String name;
  public int count;

  public SuperClass()
  {
  }

  public SuperClass(String name, int count)
  {
    this.name = name;
    this.count = count;
  }
}

public class SubClass extends SuperClass
{
  public String color;

  public SubClass(String color)
  {
    super("bubs", 5);
    this.color = color;
  }

  public SubClass(String name, int count, String color)
  {
```

```
    this.name = name;
    this.count = count;
    this.color = color;
  }

  public static void main(String[] args)
  {
    SubClass sub1 = new SubClass("red");
    SubClass sub2 = new SubClass("bimbo", 6, "blue");
    System.out.println(sub1.name + " " + sub1.count + " " + sub1.color);
    System.out.println(sub2.name + " " + sub2.count + " " + sub2.color);
  }
}
```

3. Предефиниране и препокриване на наследени методи

Предефиниране на наследени методи - добавяне на нов метод, чието име съвпада, но броя или типа на параметрите се различава от този на наследения метод.

Препокриване на метод – заместване на реализацията на метод, наследен от суперкласа с наша реализация. Името, броя и типа на връщаните параметри трябва да съвпадат.

Методът може да бъде препокрит само ако е достъпен. Ако методът не е достъпен, той не е наследен, а щом не е наследен, той не може да бъде препокрит. Например private метод не е достъпен извън собствения си клас. Ако субклас дефинира метод, който случайно има същата сигнатура и връщан тип, то те нямат нищо общо – методът на субкласа не препокрива метода на суперкласа.

4. Скриване на полета

Полетата не могат да бъдат препокривани, те могат само се скрият. Ако в субкласа декларирате поле с име, съвпадащо с името на поле от суперкласа, то полето на суперкласа все още съществува, но не е достъпно директно само със своето име. В този случай трябва да се използва или super или друга референция на суперкласа, за да имаме достъп до това поле.

Статичните членове, независимо дали методи или полета не могат да бъдат препокривани, те винаги са скрити. Но фактът, че са скрити няма голям ефект, и без това достъпът до статично поле или метод винаги трябва да става чрез името на декларалия го клас.

```
public class SuperShow
{
  public String str = "superStr";

  public void show()
  {
    System.out.println("SuperShow " + str);
  }
```

```
}

public class ExtendShow extends SuperShow
{
  public String str = "extendStr";

  public void show()
  {
    System.out.println("ExtendShow " + str);
  }

  public void test()
  {
    System.out.println("Do nothing");
  }

  public static void main(String[] args)
  {
    ExtendShow ext = new ExtendShow();
    SuperShow sup = ext;
    sup.show();
    ext.show();
    System.out.println("sup.str " + sup.str);
    System.out.println("ext.str " + ext.str);
  }
}
```

5. Ключовата дума super

Ключовата дума super е налице във всички нестатични методи на клас. При обръщане към поле или викане на метод, super служи като референция към текущия обект в ролята му на екземпляр на своя суперклас. Употребата на super е единственият случай, когато типът на референцията определя коя реализация на метода да бъде използвана. Извикването super.метод винаги използва реализацията на метод, която би използвал суперкласът. Той не използва никоя припокриваща реализация на метода от някой субклас.

Пример:

```
public class That
{
  protected String getClassName()
  {
    return "That";
  }
}

package classespackage;

public class More extends That
{
```

```
  protected String getClassName()
  {
    return "More";
  }

  public void printClassName()
  {
    That sref = (That) this;
    System.out.println("this.getClassName() = " + this.getClassName());
    System.out.println("sref.getClassName() = " + sref.getClassName());
    System.out.println("super.getClassName() = " + super.getClassName());
  }

  public static void main(String[] args)
  {
    More mref = new More();
    mref.printClassName();
  }
}
```

Макар че и двете – serf и super сочат към един и същ обект и са от тип That, само super ще игнорира истинския клас на обекта и ще използва реализацията на метода от суперкласа. Референцията serf ще работи както работи this и ще избере реализацията на метода на базата на истинския клас на обекта.

6. Класът Object.

Класът Object е базовият клас в Java. Този клас се наследява (пряко или непряко) от всички останали класове. Това означава, че всеки обект може да бъде представен като обект на класа Object. Следователно може да използва достъпните му методи. Класът Object поддържа някои базови методи. За наследниците му, някои от тях трябва (могат) да бъдат предефинирани:

- equals(Object)

Сравняването на променливи от прости типове става с оператора “==”. Но сравняването дали два обекта са с еднаква стойност става чрез метода equals, който връща резултат логическа стойност.

В Java присвояването на една обектна променлива на друга със знака “=” винаги е по адрес (създава се псевдоним), т.е. ако имаме

```
Car car1 = new Car(“Escort”);
Car car2 = car1;
```

променливите car1 и car2 сочат един и същи обект в паметта.

Ако променим стойността на car2.name = “ESCORT”, променя се стойността и на car1.

- finalize()

Този метод се извиква при премахване на обект от паметта. Това става автоматично чрез Garbage collector. Но ако има създадени native методи, които заделят памет тя трябва да бъде освободена в такъв предефиниран метод. В него трябва да има и извикването super.finalize(), за да се освободят и ресурсите заети от родителския клас.

Интерфейси

Интерфейсите предоставят възможност за реализиране на механизъм, подобен на множествено наследяване и са техника, позволяваща на един клас да има повече от един родител. Задават множество от методи и константи, които трябва да се наследят от друг клас или интерфейс. Един клас може да разширява само един клас и да реализира множество интерфейси.

Интерфейсите са подобни на класовете, с това, че имат атрибути и методи. Но атрибутите могат да бъдат само инициализирани константи (static и final, те всъщност са такива по подразбиране и могат да се дефинират като обикновени полета – int SILENT = 0;), защото интерфейсите не съдържат реализация, а методите трябва да бъдат само декларирани без да са описани телата им (методите неявно са абстрактни и публични). Създателят на интерфейса декларира методите, които трябва да се поддържат от класовете, разширяващи този интерфейс и декларира тяхното поведение. В интерфейса не може да се дава реализация на метода – това е задължение на класа, който реализира интерфейса, затова вместо тяло поставяме точка и запетая.

Интерфейсите се реализират от класовете с ключовата дума **implements**.

Интерфейсите могат да бъдат разширявани от други интерфейси чрез ключовата дума extends. Един интерфейс може да разширява един или повече интерфейси, като добавя нови константи или методи, които трябва да бъдат реализирани във всеки клас, реализиращ разширения интерфейс.

Супертипове на един клас са класовете, които той разширява и интерфейсите, които реализира, включително всички супертипове на тези класове и интерфейси. Затова обектът е екземпляр не само на своя клас, но и на всички негови супертипове, включително и интерфейсите вследствие на полиморфизма.

Не можем да създаваме обекти от интерфейс!

Декларация на интерфейс:

[public] interface име_на_ интерфейс [extends списък_от_интерфейси]

За да бъде използван един интерфейс той трябва да бъде описан след ключовата дума implements в декларацията на клас. **В класа трябва да бъдат предефинирани всички методи на интерфейса.** Във връзка с това, методите на интерфейсите могат да имат само спецификатор public и модификатор abstract. Ако не е указан спецификатор (friendly) интерфейсите могат да бъдат наследявани само в пакета.

Механизмът полиморфизъм е приложим и за интерфейсите. Всеки клас наследник на интерфейс има свойствата на интерфейса и може да бъде представен като обект на интерфейса. Но интерфейсите нямат конструктори и не наследяват класове (включително и класа Object). Те могат да наследяват само други интерфейси.

Разлика между абстрактните класове и интерфейсите:

- Интерфейсите предоставят форма на множествено наследяване, защото един клас може да реализира множество интерфейси. Един клас може да разшири само един клас, дори и този клас да има само абстрактни методи;
- Абстрактният клас може да има частична реализация, защитени части, статични методи и т.н., докато интерфейсите са ограничени само до публични константи и публични методи без реализация.

Пример: всички продукти на един производител имат общи черти: името на производителя и цената:

```
public interface Product
{
  static final String Maker = "My Corporation";
  public int getPrice(int id);
}

public class Jacet extends Object implements Product
{
  public int getPrice(int id)
  {
    switch(id)
   {
     case 1 : return 15;
     case 2 : return 21;
     default: return 30;
   }
  }

  public String toString()
  {
    return "Jacet";
  }
}

public class Shoe extends Object implements Product
{
  public int getPrice(int id)
  {
    return id*5;
  }

  public String toString()
  {
    return "Shoe";
  }
}

public class TestProduct
{
```

```
  public static void main(String args[])
  {
    Shoe shoe = new Shoe();
    Jacet jacet = new Jacet();
    Product products[] = {shoe, jacet};
    System.out.println("List of products:");
    for(int i = 0; i<products.length; i++)
    {
      System.out.print((i+1) + ". " + products[i] + ". Price for id=1: ");
      System.out.println(products[i].getPrice(1));
    }
  }
}
```

Задача:

1. Да се създаде интерфейс Person (Човек) с методи getName(), getWeight(), getHeight(),…
2. Клас Student, наследник на Person с private атрибути:
    - име, тегло, височина - които да се използват от методите на Person;
    - факултетен номер, масив с елементи от тип int за оценки;

и методи:

- конструктор, в който се инициализират всички атрибути;
- getFacultyNumber();
- за пресмятане на средния успех.

3. Програма, в която се задават двама студенти и се извежда информацията за тях.

```
public interface Person
{
  String getName();
  int getWeight();
  int getHeight();
}

public class Student implements Person
{
  private String name;
  private int weight;
  private int height;
  private String facultyNumber;
  private float[] ratings;

  public Student(String name, int weight, int height, String facultyNumber,
                 float[] ratings)
```

```
  {
    this.name = name;
    this.weight = weight;
    this.height = height;
    this.facultyNumber = facultyNumber;
    this.ratings = ratings;
  }

  public String getName()
  {
    return this.name;
  }

  public int getWeight()
  {
    return this.weight;
  }

  public int getHeight()
  {
    return this.height;
  }

  public String getFacultyNumber()
  {
    return this.facultyNumber;
  }

  public float getRate()
  {
    float rating = 0;
    for(int i = 0; i < ratings.length; i++)
    {
      rating += ratings[i];
    }

    rating /= ratings.length;
    return rating;
  }
}

public class TestStudent
{
  public static void main(String[] args)
  {
    float[] ratings1 = {3, 4, 5};
    float[] ratings2 = {6, 6, 5};
    Student stu1 = new Student("John", 90, 181, "6475", ratings1);
    Student stu2 = new Student("Mary", 55, 175, "6476", ratings2);
    Person students[] = {stu1, stu2};
    for (int i = 0; i < students.length; i++)
```

```
    {
      stu1 = (Student) students[i];
      System.out.println("Name: " + stu1.getName() + " Weight: " +
                              stu1.getWeight() + " Height: " +
                              stu1.getHeight() + " Faculty number: " +
                              stu1.getFacultyNumber() + " Rate: " +
                              stu1.getRate());
    }
  }
}
```

Изключения

1. Изключения в Java.

Не всеки метод може винаги да завърши успешно, т.е. при определени ситуации някои методи може да не могат да извършат, основната дейност, за която са предназначени и да не могат да върнат като стойност коректен резултат. Например, какво трябва да върне метода за изчисляване на факториел, ако се подаде параметър – отрицателно число? За обработка на такива ситуации е създадена специална технология за работа с изключения, която включва:

- Обработка на изключение;
- Създаване на изключение.

Изключенията са класове наследници на класа Exception. Когато в даден метод стане нещо лошо, той създава специален обект-изключение, който се прихваща от обхващаща клауза. Изключенията предоставят добър начин за прихващане на грешки, без да правим кода оборкан, добавяйки проверки за правилното изпълнение на всеки оператор в програмата ни. Те предоставят механизъм за директно сигнализиране на грешки, вместо използване на флагове или странични ефекти, които всеки път трябва да се проверяват. Отпада необходимостта програмистът да помни примерно кои са коректните числови стойности, връщани от някакъв метод и всеки път при неговото извикване да проверява връщаната стойност. Т.е. когато не използваме изключения има противоречие между коректността (проверка за всички грешки) и яснотата (да не се скрива логиката на кода зад многото грешки).

Едно изключение е обект със свои тип, методи и данни. Представянето на изключението като обект е полезно, защото обектът изключение може да включва данни, методи и възможност да рапортува изключения или да възстанови нормалното състояние.

Всеки наследник на класа Exception (по-точно на Throwable, който е суперклас на Exception) поддържа следните методи:

- getMessage() – връща низ с детайлна информация за изключението;
- toString() – конвертира обекта до низ, който можете да изведете на екрана;
- printStackTrace() – визуализира пътя от метода, предизвикал изключението до метода, в който е обработено (йерархията от методите, довели до изключението).

2. Обработване на изключение

Изключенията не се обработват в метода, който ги е предизвикал, а в този метод, който извиква метода, предизвикващ изключение. Това става със специален код,

даден по-долу. Парадигмата за обработване на изключения е последователност try-catch-finally.

```
try
{
  код, който може да предизвика изключения
}
catch(тип_на_изключение1  име_на_променлива1)
{
  код, изпълняващ се при получаване на изключение от клас ип_на_изключение1
}
catch(тип_на_изключение2  име_на_променлива2)
{
  код, изпълняващ се при получаване на изключение от клас
  тип_на_изключение2
}
finally
{
  код, изпълняващ се след приключването на изпълнението в try или catch
  клаузата
}
```

Try-catch блоковете може да се разполагат навсякъде в една програма, където се очаква да се появят някакви изключения.

В try блока се записва код, който очакваме да предизвика изключения – в него има извиквания на методи, които предизвикват изключения.

В catch блоковете се прихващат разнообразни изключения, които очакваме че могат да бъдат предизвикани от методите в try блока. При нужда имаме достъп до методите на съответния тип на изключението посредством съответната променлива.

Във finally блока се записва код, който трябва да се изпълни, независимо дали е възникнало изключение или не. Присъствието на този блок в кода не е задължително. Обикновено се използва да се възстанови начално състояние, да се освободят ресурси, които не са обекти, примерно локално отворени файлове.

```
public class TestExceptions
{
  public static void main(String args[])
  {
    double a[] = new double[1];
    try
    {
      a[0]=0;
      a[1]=1;
    }
    catch(ArrayIndexOutOfBoundsException e)
```

```
    {
      System.out.println("java.lang.ArrayIndexOutOfBoundsException");
      System.out.println("getMessage(): " + e.getMessage());
      System.out.println("toString(): " + e.toString());
      System.out.println("printStackTrace(): ");
      e.printStackTrace();
    }
    finally
    {
      System.out.println("End of test.");
    }
  }
}
```

В горния пример, грешното обръщение към втория (несъществуващ) елемент на масив (a[1]) предизвиква изключение от тип ArrayIndexOutOfBoundException. Променливата ‘e’ представя обекта-изключение, което даден метод е предизвикал.

Ако се използват няколко catch блока, обработката на изключението генерирано в try блока се предава на първия catch блок, чийто параметър съвпада с типа на изключението. Изпълнява се само тялото на този блок. Изпълнението на програмата продължава след последния catch блок.

Поради полиморфизма на класовете ако в поредица от catch блокове укажем, че се обработват първо изключения, стоящи по-високо в йерархията на изключенията винаги ще се изпълнява първия подходящ блок. Например, ако имаме първи блок catch(Exception e){ … }, последван от други блокове ще се изпълнява само първия, защото Exception е базов клас за изключенията.

Всички изключения могат да бъдат обработени едновременно като се укаже, че в catch блока ще се обработват изключения от тип Exception.

Някои изключения могат да не бъдат обработвани, но за други задължително трябва да бъде създадена try-catch конструкция. При вторите, програмата не може да бъде компилирана, ако не е създадена такава конструкция.

Всеки метод, който работи с методи, предизвикващи изключения трябва да ги обработи или да укаже, че само ги препредава като ги декларира в декларацията на метода с throws клаузата. Това са двата начина за обработване на изключения.

Има и комбиниран подход при обработката на изключения, при който изключението не само се обработва, но и се предава на извикващата функция. По този начин се дава възможност едно изключение да се обработва по различен начин в различни части от кода.

Пример:

```
protected void doRead() throws IOException
{
  int inChar =  System.in.read();
}
protected void myMethod()
```

```
{
  try
  {
    doRead();
  }
  catch (IOException e)
  {
    String err = e.toString();
    System.out.println(err);
  }
}
```

3. Типове изключения

В Java са дефинирани много изключения, които могат да се предизвикат в някой от методите на класовете на Java. За да знаем кое изключение трябва евентуално да бъде обработено при извикване на метод, то трябва да направим справка с документацията и да установим дали използваният от нас метод “изхвърля” изключение и какво е то. Проверяваните изключения на метод са също толкова важни, колкото и типът на връщаната стойност.

Ето някои от най-често генерираните от системата изключения и причините за генерирането им:

- ArithmeticException - грешка при изчисление (напр. деление на нула);
- ArrayIndexOutOfBoundsException - неправилно индексиране на масиви;
- ArrayStoreException - опит за запис в масив на стойност от неправилен тип;
- FileNotFoundException - обръщение към несъществуващ файл;
- IOException - обща входно-изходна грешка;
- NullPointException - обръщение към неинициализиран обект;
- NumberFormatException - невъзможно конвертиране между низове и числа;
- StringIndexOutOfBoundsException - опит за достъп до несъществуваща позиция в низ.

4. Създаване на собствено изключение

Става като наследим класа Throwable или негов субклас, по конвенция новите изключения наследяват класа Exception, който е наследник на Throwable, и предадем съобщението в конструктора на родителя. Конструкцията е следната:

```
public class NewException extends Exception
{
   public NewException(String message)
   {
        super(message);
```

```
    }
}
```

Причини за създаване на изключение:

- Добавяне на полезни данни, които да дадат разбираемо за човека описание на грешката;
- Възможността да се прихващат определено изключение и да се пропускат другите, които не го касаят.

Декларирането, че даден метод предизвиква изключение(я) става в декларацията на метода след ключовата дума **throws**. В списъка от изключения се записват имената на типовете изключения, които метода може да предизвика. Тук трябва да се опишат изключенията, които не се обработват вътре в самия метод.

Когато препокриваме наследен метод или реализираме наследен абстрактен метод, клаузата throws на препокриващия метод трябва да бъде съвместима с throws клаузата на наследения метод (независимо дали е абстрактен). Простото правило е, че препокриващият или реализиращият метод не може да добави изключения в своята throws клауза. Основанието за това правило е съвсем просто – кодът, написан да работи с оригиналния метод, няма да може да улови никакви допълнителни изключения.

Предизвикването на изключение в тялото на метод става с ключовата дума **throw** в тялото на метода.

Например, “if(……) throw Exception;”.

Пример: създаване на собствено изключение.

```
public class EmptyStringException extends Exception
{
  public EmptyStringException(String msg)
  {
    super("My exception message: " + msg);
  }
}

public class EmptyStringTest
{
  public String text;

  public EmptyStringTest(String text)
  {
    this.text = text;
  }
```

```
  public void printText() throws EmptyStringException
  {
    if (this.text.length() < 1)
    {
      EmptyStringException e = new EmptyStringException("The text is
                                                          empty!");
      throw e;
    }
    else
      System.out.println(this.text);
  }

  public static void main(String[] args)
  {
    EmptyStringTest est = new EmptyStringTest("");
    try
    {
      est.printText();
    }
    catch(EmptyStringException e)
    {
      System.out.println(e.getMessage());
      System.out.println(e.toString());
      e.printStackTrace();
    }
  }
}
```

5. Грешки

Грешките са по-сложен вид изключения. Грешките са наследници на класа Error (който е наследник на класа Throwable). С тях се описват много по-сериозни проблеми от изключенията – вътрешни грешки, с които програмата на може да се справи. Тези грешки се обработват от системата.