

# **Programare Orientata pe Obiecte**

**Sl.dr.ing. Marian Bucos**

## **Cuprins:**

### **1. Introducere in Java**

Limbajul de programare Java  
Programarea Orientata pe Obiecte  
Tehnologii Java (platforme Java)

### **2. Primii pasi in Java**

Instalarea Java SDK  
Compilarea si rulara unui program  
Primului program Java  
Documentarea programelor

### **3. Identificatori, cuvinte cheie, tipuri de date**

Utilizarea comentariilor intr-un program sursa  
Cuvinte cheie in Java  
Operatorii si precedenta lor  
Tipuri de date primitive si referinta  
Declararea variabilelor

### **4. Instructiuni Java pentru controlul executiei**

Instructiuni conditionale  
Instructiuni ciclice  
Alte instructiuni Java

### **5. Tablouri**

Crearea unui tablou  
Determinarea dimensiunii unui tablou  
Crearea unui tablou multidimensional

## **6. Clase Java**

Definirea unui clase

Utilizarea modificatorilor de vizibilitate si drepturi de acces

Declararea variabilelor si implementarea metodelor intr-o clasa

Instantierea obiectelor unei clase

Ierarhii de clase

Clase si metode abstracte

Crearea si utilizarea interfetelor

## **7. Exceptii**

Definirea exceptiilor

Categorii de exceptii

Tratarea exceptiilor folosind try ... catch ... finally

Definirea de exceptii utilizator

## **8. Operatii de intrare/iesire**

Definirea conceptului de flux de date

Fluxuri standard de intrare/iesire

Utilizarea fluxurilor de date

# 1. Introducere in Java

## 1.1 Limbajul de programare Java

Java este un limbaj de programare orientat pe obiect ce a fost dezvoltat de catre James Gosling si colegii sai de la Sun Microsystems la inceputul anilor 90.

Limbajul Java poate fi utilizat cu succes pentru a proiecta aplicatii ce ruleaza pe un singur calculator sau aplicatii ce sunt distribuite prin intermediul serverelor si clientilor intr-o retea. Deasemenea, Java poate fi utilizat pentru a dezvolta module sau apleturi pentru aplicatii Web.

Java este un limbaj facil de utilizat chiar si de programatorii neprofesionisti, prin eliminarea mostenirii multiple, a supraincarii operatorilor sau a pointerilor.

O alta caracteristica este portabilitatea, Java fiind independent de masina pe care lucreaza. Abilitatea de a compila o singura data si a rula pe diverse platforme se realizeaza prin intermediul compilatorului Java si a masinii virtuale Java. Natura limbajului Java il face ideal pentru cei care dezvolta aplicatii pe platforme diferite.

## 1.2 Programarea Orientata pe Obiecte

Limbajul Java a fost construit folosind conceptele modelului orientat pe obiecte.

Modelul orientat pe obiecte este bazat pe clase, obiecte si interactiunea dintre obiecte. Datele modelului reprezinta un set de obiecte care sunt instante ale claselor. Fiecare obiect are o identitate, o stare si un comportament. Obiectele sunt definite dupa modele, care contin variabilele pe care le vor folosi obiectele si metodele lor.

Modelul orientat pe obiecte este caracterizat de o serie de principii:

- *abstractizarea*: fiecare element al sistemului poate executa actiuni, isi poate modifica starea si poate comunica cu alte elemente fara a dezvalui facilitatile detinute;

- *polimorfismul*: reprezinta abilitatea de a procesa obiectele diferit, in functie de tipul lor; descrie situatia in care un nume se refera la doua metode diferite; in Java exista doua tipuri de polimorfism: tipul de supraincarcare si tipul de supradefinire;
- *incapsularea*: exprima proprietatea de opacitate a obiectelor cu privire la structura lor interna si la modul de implementare a metodelor;
- *mostenirea*: se refera la relatiile existente intre clase; o astfel de relatie permite construirea unei noi clase, denumita derivata, pornind de la clase existente, denumite de baza.

### 1.3 Platforme Java

Limbajul Java pune la dispozitie mai multe platforme de lucru pentru rezolvarea unor probleme din cele mai diverse domenii:

- *Java SE* (Standard Edition): reprezinta platforma standard de lucru pentru dezvoltarea de aplicatii si applet-uri; cuprinde doua componente de baza: JRE (Java Runtime Environment) si JDK (Java Development Kit);
- *Java EE* (Enterprise Edition): este standardul folosit in industrie pentru a dezvolta aplicatii Java server-side portabile, robuste, scalabile si sigure; avand la baza Java SE, Java EE furnizeaza servicii Web, componente si API-uri ce permit implementarea de arhitecturi orientate pe servicii (SOA) si de aplicatii Web 2.0;
- *Java ME* (Micro Edition): este o colectie de tehnologii si specificatii utilizate pentru a crea o platforma care corespunde cerintelor echipamentelor mobile; elementele acestei colectii pot fi combinate pentru a crea un mediu de executie specific unui anumit echipament.

## 2. Primii pasi in Java

### 2.1 Instalarea Java SDK

Cursul de fata trateaza platforma standard de lucru (Java SE) utilizata in dezvoltarea de aplicatii si applet-uri web.

Pentru dezvoltarea de programe Java veti avea nevoie pentru inceput de mediul de dezvoltare Java SE SDK (Software Development Kit), ce cuprinde o serie de unelte printre care compilator, masina virtuala, depanator. Kitul de instalare poate fi descarcat gratuit de la adresa <http://java.sun.com/javase/downloads>, iar documentatia aferenta se gaseste la adresa <http://java.sun.com/docs>. La finalizarea procesului de instalare al J2SDK, in directorul *director\_instalare\bin* pot fi identificate o serie de unelte ce sunt oferite de mediul de dezvoltare: compilatorul Java (*javac* – converteste programele sursa in cod binar Java), interpretorul Java sau masina virtuala Java (*java* – executa codul binar rezultat in urma compilarii).

Deasemenea, trebuiesc realizate o serie de configurari in ceea ce priveste variabilele sistem *CLASS\_PATH* si *PATH*, dupa cum urmeaza:

- *CLASSPATH* = *director\_instalare; .\*
- *PATH* = *%PATH%;director\_instalare/bin*

Programele Java pot fi scrise folosind de la cel mai simplu editor de text (notepad) pana la medii integrate de dezvoltarea a aplicatiilor (JBuilder, Eclipse, JCreator). Se recomanda folosirea mediului de dezvoltare JCreator, care poate fi descarcat de la adresa <http://www.jcreator.com/download>.

### 2.1 Compilarea si rularea unui program

Codul sursa, scris de programatori, poate fi salvat numai in fisiere ce au extensia *.java*. Este indicat ca fisierul sursa sa aiba acelasi nume cu clasa principala a aplicatiei.

Limbajul Java utilizeaza un compilator care converteste codul sursa de nivel inalt in cod binar Java. In urma compilarii va rezulta cate un fisier cu extensia *.class* pentru fiecare clasa din program.

Rularea unei aplicatii Java presupune apelarea interpretorului Java (masina virtuala Java – JVM) pentru fisierul *.class* corespunzator clasei principale a aplicatiei.

In cazul in care vorbim despre un applet Java vom intalni modificari numai in privinta rularii acestuia, compilarea realizandu-se similar cu aplicatiile Java. Applet-urile reprezinta aplicatii Java de mici dimensiuni ce pot fi rulate in browsere web. Pentru lucrul cu applet-uri, limbajul Java pune la dispozitia utilizatorilor o alta unealta numita *appletviewer*, utilizata in general in testare.

### 2.3 Primul program Java

Orice aplicatie Java este formata din unul sau mai multe fisiere sursa cu extensia *.java*, in care sunt definite clase. Aplicatiile Java trebuie sa contina o clasa principala, clasa care detine metoda *main()*.

Prima aplicatie va permite afisarea unui mesaj de bun venit in lumea Java.

✓ *BunaJava.java*

```
class BunaJava {  
    public static void main(String[] args) {  
        System.out.println("Bun venit in lumea Java!");  
    }  
}
```

La fel ca si in programele din C sau C++, prima data intr-un program Java se va executa metoda *main()*.

Dupa scriere, o aplicatie Java trebuie compilata folosind compilatorul *javac*. Acesta va fi apelat pentru fisierul sursa care contine clasa principala a aplicatiei.

In cazul nostru, vom apela din linie de comanda compilatorul Java pentru fisierul *BunaJava.java*:

```
javac BunaJava.java
```

Dupa compilare obtinem un fisier care contine cod binar Java si are extensia *.class*.  
Putem verifica aparitia acestui fisier dupa compilarea fisierului sursa *BunaJava.java*.

```
D:\java>dir
```

```
Volume in drive D is data
```

```
Volume Serial Number is 4066-D433
```

```
Directory of D:\java
```

```
02/01/2007 02:29 PM <DIR>      .
02/01/2007 02:29 PM <DIR>      ..
02/01/2007 02:29 PM          417 BunaJava.java
           1 File(s)        417 bytes
           2 Dir(s) 92,830,121,984 bytes free
```

```
D:\java>javac BunaJava.java
```

```
D:\java>java BunaJava
```

```
Bun venit in lumea Java!
```

Pentru a rula programul este necesara apelarea interpretorului Java, care asteapta ca si argument denumirea clasei.

```
java BunaJava
```

## 2.4 Documentarea programelor

Limbajul Java pune la dispozitia dezvoltatorilor un utilitar special (javadoc) pentru generarea de documentatie, pe baza comentariilor introduse in fisierele sursa. Documentatia rezultata este in format html si descrie clase, interfete, constructori, metode, etc.

Programatorul poate include comentarii de documentare (doc comments sau comentarii Javadoc) in codul sursa, inaintea declaratiei unei clase, interfete, metoda, constructor sau atribut. Comentariile Javadoc sunt introduse intre sirurile de caractere */\*\** si *\*/*. Textul dintr-un astfel de comentariu poate sa fie dispus pe una sau mai multe linii.

In general, un comentariu Javadoc este alcatuit dintr-o descriere principala, care incepe dupa caracterele `/**`, si o sectiune de tag-uri. Sectiunea de tag-uri incepe cu declaratia primului marcaj special (tag), marcaj ce este definit prin utilizarea caracterului `@`.

Tag-urile reprezinta cuvinte cheie dintr-un comentariu, care pot fi procesate de catre utilitarul javadoc. Exista doua tipuri de marcaje speciale: block tags si in-line tags. Pentru a fi interpretate, marcajele de tip block trebuie sa apara obligatoriu la inceputul unei linii. Marcajele in-line pot fi pozitionate oriunde in interiorul unui comentariu si au formatul `{@tag}`.

Principalele marcaje speciale sunt:

`@author`, `{@docRoot}`, `@deprecated`, `@exception`, `{@inheritDoc}`, `{@link}`, `{@linkplain}`, `@param`, `@return`, `@see`, `@serial`, `@serialData`, `@serialField`, `@since`, `@throws`, `{@value}`, `@version`.

In continuare vom comenta aplicatia BunVenit.java folosind taguri din lista de mai sus.

#### ✓ BunVenit.java

```
/**
 * Clasa BunVenit
 * exemplifica afisarea unui mesaj de bun venit
 * @author Marian Bucos
 * @version 1.0
 */
public class BunVenit {
    /**
     * Metoda mesaj()
     * @param nume sir de caractere folosit pentru transmiterea numelui
     * @return returneaza un mesaj de bun venit personalizat
     * @throws exceptions nu trateaza exceptii
     */
    static String mesaj(String nume) {
        String mesaj = "Bun venit in lumea Java " + nume + "!";
        return mesaj;
    }
}
```

```
    }  
    /**  
    * Metoda main()  
    * @param args tablou care contine parametrii din linia de comanda  
    * @throws exceptions nu trateaza exceptii  
    */  
    public static void main(String[] args) {  
        System.out.println(mesaj("Popescu Vlad"));  
    }  
}
```

Apelarea utilitarului javadoc pentru fisierul sursa se poate face utilizand sintaxa:

```
javadoc -author -version -d dir fisier.java
```

Optiunile *-author* si *-version* permit procesarea marcajelor *@author* si *@version* in momentul generarii documentatiei.

```
D:\java>javadoc -author -version -d docs BunVenit.java
```

```
Creating destination directory: "docs\"
```

```
Loading source file BunVenit.java...
```

```
Constructing Javadoc information...
```

```
Standard Doclet version 1.6.0
```

```
Building tree for all the packages and classes...
```

```
Generating docs\BunVenit.html...
```

```
Generating docs\package-frame.html...
```

```
Generating docs\package-summary.html...
```

```
Generating docs\package-tree.html...
```

```
Generating docs\constant-values.html...
```

```
Building index for all the packages and classes...
```

```
Generating docs\overview-tree.html...
```

```
Generating docs\index-all.html...
```

```
Generating docs\deprecated-list.html...
```

```
Building index for all classes...
```

```
Generating docs\allclasses-frame.html...
```

```
Generating docs\allclasses-noframe.html...
```

```
Generating docs\index.html...
```

```
Generating docs\help-doc.html...
```

```
Generating docs\stylesheet.css...
```

### 3. Identificatori, cuvinte cheie, tipuri de date

#### 3.1 Utilizarea comentariilor intr-un program sursa

Comentariile reprezinta o parte importanta a unui program. Ele permit descrierea zonelor de program pentru o mai buna intelegere in cazul unei viitoare utilizari.

Limbajul Java recunoaste trei tipuri de comentarii:

- comentarii pe mai multe linii (comentarii traditionale) – sunt delimitate de sirurile de caractere `/*, */;`
- comentarii pentru generarea de documentatie – sunt introduse prin intermediul caracterelor `/**` si se inchid prin `*/;`
- comentarii pe o singura linie (stilul C++) – acestea incep cu `//` si se termina la incheierea liniei de program; sunt utile in descrierea liniilor de program.

#### 3.2 Cuvinte cheie in Java

In Java, cuvintele cheie sunt termeni rezervati care nu pot fi utilizati ca si identificatori. Termenii *const* si *goto* sunt rezervati chiar daca nu sunt folositi in momentul de fata.

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

Identificatorii sunt expresii alese de catre programatori pentru a reprezenta nume de clase, obiecte, variabile, constante sau metode. In limbajul Java identificatorii trebuie sa inceapa cu o litera, dupa care pot urma atat litere cat si cifre. Deasemenea, in identificatori se face distinctie intre literele mari si cele mici.

### 3.3 Operatorii si precedenta lor

Operatorii sunt simboluri speciale care permit implementarea operatiilor dintre operanzi.

Dupa tipul lor operatorii Java pot fi clasificati astfel:

- operatori de asignare: = += -= \*= /= %=
- operatori relationali: == != < > <= >=
- operatori logici: && || ^ !
- operatori logici pe biti: & | ^
- operatori aritmetici: + - \* / %
- operatori de translatie: << >> >>>
- operatori incrementare: ++ --
- operator conditional: (expresie\_logica) ? valoare\_adevarat: valoare\_fals
- operator de concatenare a sirurilor: +
- operatori pentru casting: (tip\_data)

Operatorii de atribuire aritmetica ( += -= \*= /= %= ) ofera o modalitate mai simpla de atribuire a unei valori. De exemplu, urmatoarele doua linii au acelasi rezultat:

```
a = a + 1;
```

```
a += 1;
```

Java include un operator special care poate inlocui unele instructiuni conditionale if-else. Acest operator poarta numele de operator conditional si are urmatorul format:

```
expresie1 ? expresie2 : expresie3.
```

Daca in urma evaluarii *expresiei1* rezultatul este adevarat, atunci se exalueaza *expresie2* si rezultatul acesteia devine rezultatul operatiei. In caz contrar, se evalueaza *expresie3* si rezultatul acesteia devine rezultatul operatiei.

## ✓ OpCond.java

```

class OpCond {
    public static void main(String[] args) {
        int varsta = 15;
        String tipPersoana;
        tipPersoana = (varsta < 18) ? "adolescent": "tanar";
        System.out.println("Varsta " + varsta + " ani corespunde unui " + tipPersoana);
    }
}

```

In evaluarea expresiilor un rol important poarta precedenta operatorilor. Operatorii cu precedenta mai mare sunt evaluati inaintea operatorilor cu precedenta mai scazuta.

()	[]	.	
++	--	!	
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
&&			
?:			
=	op=		

Operatia de casting, sau schimbare de tip, presupune atribuirea unei valori de un tip la o variabila de un alt tip. Schimbare de tip se poate realiza implicit, prin intermediul compilatorului, sau explicit, in cazul in care utilizatorul precizeaza conversia dorita. Trebuie avut in vedere faptul ca aceasta conversie poate provoca pierderea preciziei.

## ✓ Casting.java

```

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

```

```
//conversie implicita
double media;
int a = 43;
short b = 15;
media = (a + b)/2;
System.out.println("Media = " + media);
//conversie explicita
char c = 'a';
int d = 99;
System.out.println("Conversie de la char la int: " + (int)c );
System.out.println("conversie tip de la int la char: " + (char)d );
}
}
```

### 3.4 Tipurile de date primitive si referinta

Java pune la dispozitie programatorilor opt tipuri de date primitive. Un tip de date primitiv este predefinit, iar denumirea sa este un cuvânt rezervat.

Cele opt tipuri de date primitive suportate de limbajul de programare Java sunt:

- byte: numere întregi pe 8 biti, având valori între -128 și 127;
- short: numere întregi pe 16 biti, având valori între -32.768 și 32.767;
- int: numere întregi pe 32 biti, având valori între -2.147.483.648 și 2.147.483.647;
- long: numere întregi pe 64 biti, având valori cuprinse între  $-2^{63}$  și  $2^{63}-1$ ;
- float: numere în virgulă mobilă și simplă precizie pe 32 biti;
- double: numere în virgulă mobilă și dublă precizie pe 64biti;
- boolean: true, false;
- char: caractere UNICODE pe 16 biti.

În afara tipurilor de date primitive, Java oferă suport special pentru șirurile de caractere prin intermediul clasei *java.lang.String*.

Tipuri de date speciale sunt referințele; din această categorie fac parte vectorii, clasele și interfețele. O variabilă de acest tip este o adresă de memorie către o valoare sau multe de valori.

### 3.5 Declararea variabilelor

O variabila are un nume, un tip, o valoare. Declararea unei variabile specifica numele si tipul variabilei, optional poate specifica si o valoare initiala.

In limbajul Java sunt definite patru tipuri de variabile:

- variabile instanta (attribute non-stactice): starea obiectelor este salvata in astfel de variabile; se numesc variabile instanta deoarece valorile lor sunt unice pentru fiecare instanta a clasei;
- variabile clasa (attribute statice): sunt variabile ce se declara prin intermediul modifierului *static*; variabilele clasa detin o valoare unitara pentru toate instancele clasei;
- variabile locale: reprezinta variabilele declarate in interiorul unei metode;
- parametrii.

Pentru a defini variabile finale sau constante trebuie utilizat modifierul *final* inaintea denumirii constantei.

## 4. Instructiuni Java pentru controlul executiei

Limbajele de programare utilizeaza instructiuni de control pentru a ajusta modul in care un program se desfasoara. Instructiunile de control din Java pot fi impartite in trei categorii:

- instructiuni conditionale;
- instructiuni iterative;
- instructiuni de salt.

### 4.1 Instructiuni conditionale

Instructiunile conditionale permit executia conditionata a unei instructiuni sau a unui bloc de instructiuni. Java suporta doua astfel de instructiuni: *if* si *switch*.

Instructiunea **IF** poate fi utilizata pentru a indruma executia programului pe doua cai.

Sintaxa generala a instructiunii este:

```
if (conditie) instructiune1;  
else instructiune2;
```

In acest caz *instructiune* poate reprezenta o singura instructiune sau un bloc de instructiuni. Daca conditia este adevarata, atunci se executa instructiune1, in caz contrar se executa instructiune2. Ramura *else* intr-o astfel de instructiune conditionala nu este obligatorie.

Instructiunea *if* poate fi intalnita si in constructii imbricate, de genul *if – else – if*.

```
if (conditie1) instructiune1;  
else if (conditie2) instructiune2;  
    else ...
```

Instructiunea **SWITCH** este utilizata pentru evaluarea valorilor posibile ale unei expresii. Expresia evaluata trebuie sa fie un intreg, deci pot fi utilizate tipurile: *byte*, *short*, *long*, *char* sau *int*.

```
switch(expresie) {  
    case valoare1: instructiune1; break;  
    case valoare2: instructiune2; break;  
    ...  
    case valoareN: instructiuneN; break;
```

```
    default: instructiune default; break;
}
```

Daca expresie are aceeasi valoare cu cea intalnita pe o ramura *case*, atunci controlul este transmis catre ramura respectiva. Vor fi rulate toate instructiunile care se gasesc inaintea primului *break*. Daca nici una din valorile intalnite nu corespunde, controlul este acordat etichetei *default* (daca aceasta exista).

### ✓ Conditie.java

```
class Conditie {
    public static void main(String[] args) {
        // instructiunea if
        int min;
        int a = 15, b = 3, c = 23;
        if (a < b) { if (a < c) min = a;
                    else min = c; }
        else { if (b < c) min = b;
                else min = c; }
        System.out.println("Minimul este: " + min);
        // instructiunea switch
        int luna = 5;
        switch (luna) {
            case 1:
            case 2:
            case 12: System.out.println("iarna"); break;
            case 3:
            case 4:
            case 5: System.out.println("primavara"); break;
            case 6:
            case 7:
            case 8: System.out.println("vara"); break;
            case 9:
            case 10:
            case 11: System.out.println("toamna"); break;
        }
    }
}
```

## 4.2 Instructiuni iterative

Instructiunile iterative utilizate in limbajul de programare Java sunt *while*, *do-while* si *for*. Aceste instructiuni au rolul de a executa repetitiv o instructiune sau un set de instructiuni pana la indeplinirea unei *conditii de iesire*.

Instructiunea iterativa *while* se utilizeaza pentru a repeta o instructiune sau un bloc de instructiuni atata timp cat conditia este adevarata.

Forma generala a instructiunii *while* este urmatoarea:

```
while (conditie) {  
    instructiune;  
}
```

In momentul in care conditia devine falsa, controlul executiei este acordat urmatoarei linii de program de dupa bucla *while*.

Instructiunea *while* mai poarta si denumirea de bucla cu test initial.

Daca se doreste testarea conditiei buclei, la finalul acesteia, se poate utiliza instructiunea *do-while*. In acest caz corpul buclei se executa cel putin o data, deoarece conditia se testeaza la final.

Sintaxa pentru *do-while* are forma:

```
do {  
    instructiune;  
} while (conditie);
```

Cea de-a treia instructiune iterativa este *for*, sau bucla cu numar fix de pasi.

Sintaxa completa a unui ciclu *for* este:

```
for (initializare; conditie; iteratie) {  
    instructiune;  
}
```

Modul de executie pentru un ciclu *for* este urmatorul: se ruleaza mai intai initializare, apoi se testeaza conditie. Daca conditia este adevarata se executa corpul ciclului si se trece la iteratie.

✓ Iteratie.java

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

```
int i;
// instructiunea while
i = 0;
while (i<10) {
    System.out.println(i);
    i++;
}
// instructiunea do-while
i = 0;
do {
    System.out.println(i);
    i++;
} while (i<10);
// instructiune for
for ( i=0; i<10; i++)
    System.out.println(i);
}
}
```

### 4.3 Alte instructiuni Java

Pe langa instructiunile conditionale si iterative, limbajul Java mai ofera trei tipuri de instructiuni de salt: `break`, `continue` si `return`. Instructiunile de salt sunt utilizate pentru a transfera controlul executiei unei alte zone a programului.

Instructiunea `break` poate fi utilizata in trei situatii: terminarea unei instructiuni `switch`, parasirea unei bucle si salt la o eticheta.

#### ✓ InstrBreak.java

```
class InstrBreak {
    public static void main(String[] args) {
        int i;
        i = 0;
        while (i < 10) {
            if (i == 5) break;
            System.out.println(i);
            i++;
        }
    }
}
```

```
    }  
    out:  
    for (i=0; i<10; i++) {  
        if (i == 5) break out;  
        System.out.println(i);  
    }  
}  
}
```

O instructiune *continue* poate aparea numai in interiorul unui bloc de instructiuni subordonat unei instructiuni iterative. Continue opreste executia restului de instructiuni din bloc si face un salt la urmatoarea iteratie. Si in cazul instructiunii continue poate fi definita o eticheta pentru salt.

✓ InstrContinue.java

```
class InstrContinue {  
    public static void main(String[] args) {  
        int i;  
        for (i=0; i<10; i++) {  
            System.out.println(i);  
            if (i%2 == 0) continue;  
            System.out.println(" ");  
        }  
    }  
}
```

Instructiunea *return* ofera controlul executiei apelantului metodei care o contine. Deasemenea, instructiunea return va determina finalizarea metodei in care este executata.

## 5. Tablouri

Tablourile reprezinta structuri de lungime fixa ce pot stoca valori de acelasi tip. In limbajul Java tablourile extind clasa `java.lang.Object`. Valorile retinute in elementele unui tablou pot fi date primitive sau referinte. Fiecare element dintr-un tablou este identificat unic de un indice numeric.

### 5.1 Crearea unui tablou

Un tablou este declarat in acelasi fel in care este declarata o variabila obisnuita. Tabloul trebuie sa detina un tip si un identificator valid. Tipul tabloului este dat de tipul elementelor continute in tablou.

Declararea unui tablou se poate face in doua moduri:

```
tip_tablou [ ] nume_tablou;
```

```
tip_tablou nume_tablou [ ];
```

Dupa declararea unui tablou, poate fi alocata memorie pentru elementele lui. Instantierea unui tablou se realizeaza prin intermediul operatorului *new*.

```
nume_tablou = new tip_tablou [numar_elemente];
```

Este posibil ca atat declararea, cat si instantierea tabloului sa se realizeze intr-un singur pas:

```
tip_tablou nume_tablou [ ] = new tip_tablou [numar_elemente];
```

```
tip_tablou nume_tablou [ ] = {e1, e2, ... , eN};
```

Referirea unui element al tabloului se realizeaza prin intermediul operatorului `[ ]`. Acesta primeste o valoare intreaga si returneaza elementul din tablou corespunzator aceluia index. Primul element dintr-un tablou are asociat indexul zero.

### 5.2 Determinarea dimensiunii unui tablou

Fiecare tablou are un atribut *length* care specifica numarul de elemente din tablou.

```
numar_elemente = nume_tablou.length;
```

✓ `Tablou.java`

```
class Tablou {
```

```

public static void main(String[] args) {
    String culori[] = {"alb", "verde", "rosu", "gaben", "albastru", "negru"};
    int i;
    for (i=0; i<culori.length; i++) {
        System.out.println(culori[i]);
    }
}

```

O alta modalitate de parcurgere a elementelor tabloului, mult mai eficienta, este urmatoarea:

#### ✓ Tab.java

```

class Tab {
    public static void main(String[] args) {
        String culori[] = {"alb", "verde", "rosu", "gaben", "albastru", "negru"};
        for (String culoare: culori) {
            System.out.println(culoare);
        }
    }
}

```

Aplicatia urmatoare implementeaza un algoritm simplu de sortare a elementelor unui tablou de numere intregi generate aleator cu ajutorul metodei Math.random(). Metoda Math.random() genereaza numere reale cuprinse intre 0.0 si 1.0. Pentru ca elementele tabloului sa fie numere intregi, valoarea generata este trecuta la tipul int prin intermediul unei operatii de casting (schimbare de tip).

```
(int)(Math.random() * 100);
```

#### ✓ Sortare.java

```

class Sortare {
    public static void sortare(int [] numere) {
        for(int i =0; i < numere.length; i++) {
            int min = i;
            for(int j = i; j < numere.length; j++) {
                if (numere[j] < numere[min]) min = j;
            }
        }
    }
}

```

```

        }
        int temp;
        temp = numere[i];
        numere[i] = numere[min];
        numere[min] = temp;
    }
}
public static void main(String[] args) {
    int [] numere = new int[10];
    for(int i = 0; i < numere.length; i++) {
        numere[i] = (int)(Math.random() * 100);
    }
    System.out.println("Afisare elemente tablou inainte de sortare");
    for (int i = 0; i < numere.length; i++) {
        System.out.println(numere[i]);
    }
    sortare(numere);
    System.out.println("Afisare elemente tablou dupa sortare");
    for (int i = 0; i < numere.length; i++) {
        System.out.println(numere[i]);
    }
}
}

```

Marginile unui tablou sunt intotdeauna verificate in limbajul Java. Daca indexul asociat unui element al tabloului este mai mic decat 0 sau mai mare sau egal cu lungimea tabloului, atunci va fi generata o eroare de genul *ArrayIndexOutOfBoundsException*, iar programul va fi incheiat.

### 5.3 Crearea unui tablou multidimensional

In Java exista posibilitatea de a declara tablouri multidimensionale, tablouri ale caror elemente sunt tot tablouri.

De exemplu, declararea si instantierea unei matrici se realizeaza astfel:

```

tip_matrice nume_matrice [ ] [ ] = new tip_matrice [numar_linii] [numar_coloane];
tip_matrice [ ] [ ] nume_matrice = new tip_matrice [numar_linii] [numar_coloane];

```

Parcurea unui tablou multidimensional se realizeaza folosind cicluri imbricate. De exemplu, pentru a parcurge o matrice sunt necesare doua bucle *for*. Bucla interioara se executa in intregime pentru fiecare iteratie a buclei exteriere.

✓ Matrice.java

```
class Matrice {  
    public static void main(String[] args) {  
        int [][] matrice = { {12, 4, 23, 1},  
                             {16, 8, 0, 21},  
                             {1, 13, 2, 17} };  
        for (int i = 0; i < matrice.length; i++) {  
            for (int j = 0; j < matrice[i].length; j++) {  
                System.out.println("matrice["+i+", "+j+"]="+ matrice[i][j]);  
            }  
            System.out.println(" ");  
        }  
    }  
}
```

## 6. Clase Java

### 6.1 Definirea unei clase

Clasele sunt elemente fundamentale ale oricarei aplicatii Java. Ele reprezinta structuri logice care definesc comportamentul si starea obiectelor.

O clasa este definita pentru a descrie un nou tip de date. Dupa definirea clasei, aceasta poate fi utilizata pentru a crea obiecte de tipul respectiv. Deci, o clasa reprezinta un sablon pentru un obiect, iar un obiect reprezinta o instanta a unei clase.

Forma generala pentru declararea unei clase este:

```
[modificatori_clasa] class nume_clasa [extends super_clasa][implements lista_interfete] {  
    // corp_clasa  
}
```

Dupa cum se poate observa, sintaxa de mai sus cuprinde o serie de termeni optionali:

- *modificatori\_clasa*: definesc aria de vizibilitate si drepturile de acces, precum si o serie de proprietati; pentru o clasa pot fi utilizati ca si modificatori: public, abstract si final;
- *extends super\_clasa*: precizeaza clasa de baza (superclasa) pentru clasa definita; Java permite doar mostenirea simpla, ceea ce inseamna ca o clasa poate avea doar un singur parinte
- *implements lista\_interfete*: listeaza interfetele ce sunt implementate de noua clasa.

Corpul unei clase poate contine doua tipuri de elemente: date membre, mai sunt denumite atribute, si functii membre, denumite si metode. Atributele si metodele definite intr-o clasa reprezinta membrii clasei.

### 6.2 Utilizarea modifcatorilor de vizibilitate si drepturi de acces

Modificatorii sunt cuvinte cheie care precizeaza aria de vizibilitate si drepturile de acces ale clientilor la membrii unei clase. Clienti pot fi metodele sau clasele care acceseaza din exterior un membru.

Modificatorii au o arie larga de utilizare care poate cuprinde definitii de clase, atribute sau metode. Urmatorii termeni pot fi utilizati ca si modificatori de vizibilitate: *public*, *protected*, *private*.

Daca nu este precizat un modificador de vizibilitate pentru un membru, atunci acesta este accesibil la nivelul pachetului din care face parte clasa in care acesta se gaseste.

Modificadorul *public* utilizat in cadrul declaratiei unei clase face ca aceasta sa fie vizibila de oriunde. Daca modificadorul public precede declararea unei variabile sau a unei metode, aceasta este vizibila oriunde este vizibila clasa.

```
public int variabilaPublica;
```

La nivelul membrilor unei clase poate fi utilizat modificadorul *private*, care stabileste vizibilitatea membrilor numai in interiorul clasei in care sunt declarati.

```
private int variabilaPrivata;
```

Modificadorul *protected* este utilizat in declararea variabilelor sau a metodelor. Membrii declarati cu *protected* sunt accesibili in cadrul clasei, subclaselor sau pachetului din care face parte clasa.

```
protected int metodaProtejata();
```

Pentru declararea variabilelor de clasa (atribute statice) se poate utiliza un modificador special, *static*. Variabilele declarate cu *static* sunt partajate de catre obiectele clasei. Daca modificadorul *static* apare in declaratia unei metode, vorbim despre metode de clasa, metode ce pot fi apelate si fara a instantia obiecte ale clasei.

```
public static int valoarePartajata;
```

Un alt modificador care este utilizat in cazul membrilor unei clase este *final*. Variabilele declarate cu *final* au o valoare constanta ce nu poate fi modificata prin program. Atunci cand o metoda este declarata ca finala nici o alta clasa nu va putea supradefini metoda, adica nu va putea fi redefinita intr-o subclasa.

```
public static final int MAX = 256;
```

O clasa care contine metode neimplementate poarta denumirea de clasa abstracta, si se declara utilizand modificatorul *abstract*. Metodele abstracte (neimplementate) continute de o clasa abstracta se declara folosind modificatorul *abstract*.

```
public abstract double aria();
```

Pe baza modificatorilor de vizibilitate pot fi definite patru niveluri de acces: *public*, *private*, *protected*, *package* (implicit).

## 6.2 Declararea variabilelor si implementarea metodelor intr-o clasa

In limbajul Java orice clasa contine doua tipuri de elemente: atribute si metode.

Atributele reprezinta aspecte individuale care diferentiaza un obiect de altul, si determina modul de afisare, starea sau calitatile unui obiect. Atributele din interiorul unei clase pot fi: *statice* si *non-statice*.

*Atributele statice* sunt asociate clasei, si sunt partajate de toate obiectele clasei. Aceste atribute pot fi utilizate chiar si in situatia in care nu se instantiaza obiecte ale clasei. Se mai numesc si *variabile de clasa*, deoarece aceste atribute apartin clasei si nu unui obiect anume. Pentru a declara o variabila de clasa trebuie utilizat modificatorul *static*.

*Atributele non-statice* sau *variabilele instantia* reprezinta cel de-al doilea tip de atribute dintr-o clasa. Fiecare instantia a unei clase va detine o copie a variabilelor instantia care sunt definite in respectiva clasa. Deasemenea, fiecare obiect va avea propriile valori pentru fiecare atribut non-static.

✓ Cerc.java

```
class Cerc {  
    //variabile_clasa  
    static final double PI = 3.14;  
    //variabile_instanta  
    double x;  
    double y;  
    double raza;  
}
```

Dupa cum se poate observa declararea unei variabile are urmatoarea forma:

```
[modificatori] tip_variabila nume_variabila [= valoare_inicializare];
```

In functie de tipul modificatorilor prezenti in declaratia unui atribut se poate preciza nivelul de acces la acel atribut, tipul atributului (instanta, clasa) sau se poate stabili daca atributul este o constanta (variabila finala).

Metodele unei clase definesc comportamentul clasei respective. In mod similar atributelor, intr-o clasa pot fi definite doua tipuri de metode: *stative* sau *metode clasa* si *non-stative* sau *metode instanta*.

Metodele statice se declara utilizand modificatorul static si pot fi apelate chiar si in cazul in care nu au fost instantiate obiecte ale clasei. Deoarece acest tip de metode pot fi executate si in absenta obiectelor, ele nu pot referi variabile instanta. Metoda main() trebuie declarata tot timpul folosind modificatorul static.

✓ Cerc.java

```
class Cerc {  
    static final double PI = 3.14;  
    static int count = 0;  
    double x;  
    double y;  
    double raza;  
    //metoda non-statica  
    public double aria() {  
        return PI*raza*raza;  
    }  
    //metoda statica  
    static int nrObiecte() {  
        return count;  
    }  
}
```

Declararea unei metode se face respectand urmatoarea sintaxa generala:

```
[modificatori] tip_metoda nume_metoda( [lista parametrii] ) [throws tip_clasa1, ... ] {
```

```
//corp metoda  
}
```

Clauza `throws` este utilizata pentru declararea exceptiilor care pot rezulta in urma executiei unei metode. Exceptiile reprezinta conditii care apar la rulare si necesita oprirea programului, fiind tratate prin obiecte ale casei `java.lang.Throwable`.

Unul din principiile de baza in programarea orientata pe obiecte este polimorfismul. Polimorfismul se refera la situatia in care un nume se refera la doua metode diferite. Limbajul Java permite definirea a doua tipuri de polimorfism: tipul de supraincarcare si tipul de supradefinire.

Polimorfismul de supraincarcare apare la definirea de metode cu acelasi nume in cadrul unei clase. Diferenta intre metode se face prin numarul de parametrii sau tipurile diferite ale acestor parametrii. Nu poate fi realizata supraincarcarea prin utilizarea de tipuri returnate diferite pentru metode.

#### ✓ Carte.java

```
class Carte {  
    String titlu;  
    String autor;  
    Carte() {  
        titlu = " ";  
        autor = " ";  
    }  
    Carte(String titlu, String autor) {  
        this.titlu = titlu;  
        this.autor = autor;  
    }  
    void fisaCarte() {  
        System.out.println("Titlu: " + titlu);  
        System.out.println("Autor: " + autor);  
    }  
    void fisaCarte(int id) {  
        System.out.println("Id: " + id);  
    }  
}
```

```

        System.out.println("Titlu: " + titlu);
        System.out.println("Autor: " + autor);
    }
    public static void main(String [] args) {
        Carte c1 = new Carte("Morometii","Marin Preda");
        Carte c2 = new Carte("Fratii Jderi","Mihail Sadoveanu");
        c1.fisaCarte();
        c1.fisaCarte(1);
        c2.fisaCarte(2);
    }
}

```

Cuvantul cheie *this* returneaza o referinta catre obiectul curent. In interiorul metodei constructor pentru clasa *Carte*, *this* este utilizat pentru a face diferenta intre variabilele de instanta si variabilele locale (de metoda), deoarece acestea au aceiasi denumire.

```
this.titlu = titlu;
```

Polimorfismul de supradefinirea apare in momentul in care metoda unei clase are acelasi nume si aceeasi semnatura cu o metoda din clasa de baza. Semnatura unei metode este data de numarul parametrilor, tipul si ordinea acestora. Spunem ca metoda din clasa derivata supradefineste metoda din clasa de baza daca numele si semnaturile metodelor coincid.

#### ✓ Firma.java

```

class Firma {
    public String denumire;
    public int angajati;
    private int departamente;
    Firma() {
        denumire = " ";
        angajati = 0;
        departamente = 0;
    }
    Firma (String denumire) {
        this();
    }
}

```

```

        this.denumire = denumire;
    }
    void setDepartamente(int departamente) {
        this.departamente = departamente;
    }
    void setAngajati(int angajati) {
        this.angajati = angajati;
    }
    void vizualizareDate() {
        System.out.println("Denumire firma: " + denumire);
        System.out.println("Numar angajati: " + angajati);
    }
}

```

Clasa *Departament* introduce prin intermediul polimorfismului metoda *vizualizareDate()*. Aceasta suprascrie metoda din superclasa *Firma* care are acelasi nume si aceiasi semnatura.

#### ✓ Departament.java

```

class Departament extends Firma {
    String nume;
    int angajati;
    Departament () {
        super();
        nume = " ";
        angajati = 0;
    }
    Departament (String nume, String denumire) {
        super(denumire);
        this.nume = nume;
    }
    void setAngajati(int angajati, int ang) {
        super.angajati = ang;
        this.angajati = angajati;
    }
    void vizualizareDate() {

```

```

        super.vizualizareDate();
        System.out.println("Nume departament: " + nume);
        System.out.println("Numar angajati departament: " + angajati);
    }
}

```

Cuvantul cheie *super* permite accesarea atributelor si metodelor superclasei. Astfel, in constructorul din subclasa se poate accesa constructorul din clasa de baza.

✓ Aplicatie.java

```

class Aplicatie {
    public static void main(String [] args) {
        Departament d1 = new Departament("Achizitii", "Continental");
        d1.setAngajati(12,259);
        d1.vizualizareDate();
    }
}

```

## 6.4 Instantierea obiectelor unei clase

Dupa cum am declarat mai sus, definirea unei clase inseamna crearea unui nou tip de date. Acesta poate fi utilizat pentru declararea de obiecte de acest tip. Crearea obiectelor se realizeaza prin instantierea unei clase si presupune doua etape: declarare si instantiere.

Prima etapa presupune declararea tipului de data pentru obiectul nou creat.

```
nume_clasa nume_obiect;
```

Cea de-a doua etapa este realizata prin intermediul operatorului de instantiere *new*. Operatorul *new* aloca dinamic memorie pentru obiectul creat si returneaza o referinta catre acesta. Tot in aceasta etapa este apelat unul din constructorii clasei, care initializeaza starea unui obiect imediat dupa crearea sa.

```
nume_obiect = new constructor();
```

Crearea unui obiect poate fi realizata si intr-o singura linie de cod de genul:

```
nume_clasa nume_obiect = new constructor();
```

Constructorul este un tip special de metoda, care are acelasi nume ca si clasa in care este definita si care nu returneaza nimic. Metodele constructor sunt apelate in momentul in care sunt create instante ale clasei. Daca nu este definita o metoda constructor pentru o clasa, compilatorul va crea o astfel de metoda. Constructorul implicit nu are lista de parametrii, iar corpul metodei este vid. Rolul principal al constructorului este acela de a realiza initializarea variabilelor instanta ale unui obiect, cand acesta este creat.

✓ Cerc.java

```
class Cerc {
    static final double PI = 3.14;
    static int count = 0;
    int x;
    int y;
    int raza;
    //metoda constructor 1
    Cerc() {
        x = 0;
        y = 0;
        raza = 0;
    }
    //metoda constructor 2
    Cerc(int x0, int y0, int raza0) {
        x = x0;
        y = y0;
        raza = raza0;
    }
    //metoda non-statica
    public double aria() {
        return PI*raza*raza;
    }
    //metoda statica
    static int nrObiecte() {
        return count;
    }
    public static void main(String[] args) {
        Cerc c = new Cerc(2, 5, 4);
    }
}
```

```
    }  
}
```

Referirea atributelor si metodelor pentru un obiect se face astfel:

```
nume_obiect.atribut;  
nume_obiect.nume_metoda();
```

Limbajul Java permite definirea de constructori multipli intr-o clasa, prin intermediul polimorfismului.

De exemplu, intr-o clasa care defineste numerele complexe, putem utiliza un constructor pentru a initializa partea reala si partea imaginara cu valoarea 0.0, si un alt constructor care permite utilizatorului sa specifice valorile initiale pentru partea reala, respectiv imaginara.

✓ Complex.java

```
class Complex {  
    Complex() {  
        parteaReala = 0.0;  
        parteaimaginara = 0.0;  
    }  
    Complex(double real, double imag) {  
        parteaReala = real;  
        parteaimaginara = imag;  
    }  
    private double parteaReala;  
    private double parteaimaginara;  
}
```

## 6.5 Ierarhii de clase

In programarea orientata pe obiecte, mostenirea permite construirea de ierarhii de clase. Mostenirea reprezinta o proprietate a claselor prin intermediul careia poate fi definita o noua clasa, care sa preia attributele si metodele unei clase mai vechi. Aceasta proprietate este implementata cu ajutorul claselor derivate.

Clasa derivata, sau subclasa, se gaseste intotdeauna pe un nivel inferior celui corespunzator superclasei, sau clasei de baza.

In limbajul Java poate fi utilizata numai mostenirea simpla, adica fiecare clasa derivata are o singura clasa de baza (parinte).

Relatia de derivare este precizata la definirea clasei derivate, folosind constructia *extends*:

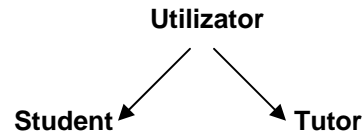
```
[modificatori_clasa] class nume_clasa [extends super_clasa][implements lista_interfete] {  
    // corp_clasa  
}
```

In general, cu cat ne deplasam mai mult de la varful ierarhiei de clase catre margini, clasele devin tot mai specializate.

#### ✓ Utilizator.java

```
class Utilizator {  
    int idUtilizator;  
    String numeUtilizator;  
    String parolaUtilizator;  
    Utilizator() {  
        idUtilizator = 0;  
        numeUtilizator = " ";  
        parolaUtilizator = " ";  
    }  
    Utilizator(int id, String nume, String parola) {  
        idUtilizator = id;  
        numeUtilizator = nume;  
        parolaUtilizator = parola;  
    }  
    void afisareInformatii() {  
        System.out.println("Id utilizator: " + idUtilizator);  
        System.out.println("Nume utilizator: " + numeUtilizator);  
        System.out.println("Parola utilizator: " + parolaUtilizator);  
    }  
}
```

Prin derivarea clasei *Utilizator* pot fi construite noi clase, *Student*, *Tutor*, care mostenesc caracteristicile clasei de baza. Aceste clase introduc insa si caracteristici noi, specifice modelelor definite.



✓ Student.java

```
class Student extends Utilizator {
    double medieStudent;
    Student() {
        super();
        medieStudent = 0.0;
    }
    Student(int id, String nume, String parola, double medie) {
        super(id, nume, parola);
        medieStudent = medie;
    }
    void afisareInformatii() {
        super.afisareInformatii();
        System.out.println("Medie: " + medieStudent);
    }
}
```

✓ Tutor.java

```
class Tutor extends Utilizator {
    int numarDiscipline;
    Tutor() {
        super();
        numarDiscipline = 0;
    }
    Tutor(int id, String nume, String parola, int discipline) {
        super(id, nume, parola);
        numarDiscipline = discipline;
    }
    void afisareInformatii() {
```

```
        super.afisareInformatii();
        System.out.println("Numar discipline: " + numarDiscipline);
    }
}
```

✓ Aplicatie.java

```
class Aplicatie {
    public static void main(String [ ] args) {
        Student s1 = new Student(1, "stud1", "parola1", 7.8);
        Student s2 = new Student(2, "stud2", "parola2", 5.9);
        Tutor t1 = new Tutor(3, "tutor1", "parola3", 4);
        s1.afisareInformatii();
        s1.afisareInformatii();
        t1.afisareInformatii();
    }
}
```

In cazul in care nu se doreste extinderea unei clase, sau supradefinirea metodelor acesteia, este necesara marcarea clasei la definire cu modificatorul final. O metoda declarata cu modificatorul final nu poate fi supradefinita.

Toate clasele in Java sunt derivate din aceiasi superclasa: Object. Aceasta clasa este definita in pachetul *java.lang* si reprezinta varful ierarhiei de clase Java. Din acest motiv, toate clasele mostenesc metodele care sunt definite in clasa Object.

Printre cele mai importante metode din clasa Object sunt:

- equals() - este utilizata pentru a compara doua obiecte;
- toString() - permite asocierea de informatie de tip String fiecarui obiect.

## 6.6 Clase si metode abstracte

Clasele abstracte sunt utilizate pentru a declara caracteristici comune unor subclase. O clasa abstracta nu poate fi instantiata. Ea poate fi utilizata numai ca si superclasa pentru alte clase care extind clasa abstracta. Clasele abstracte sunt declarate prin intermediul modifierului abstract.

O clasa abstracta poate contine atribute, ce descriu caracteristicile clasei, si metode, care descriu actiunile ce pot fi desfasurate de clasa. Deasemenea, o clasa abstracta poate include metode care nu sunt implementate. Aceste metode au numai declaratie si poarta numele de metode abstracte. Ca si in cazul claselor abstracte, metodele abstracte sunt insotite in declaratia lor de modificatorul abstract.

Daca o clasa contine metode abstracte, atunci clasa trebuie sa fie declarata abstracta:

✓ `ObiectGrafic.java`

```
public abstract class ObiectGrafic {
    protected double x, y;
    public ObiectGrafic() {
        x = 0;
        y = 0;
    }
    public ObiectGrafic(double x0, double y0) {
        x = x0;
        y = y0;
    }
    public String toString() {
        return "ObiectGrafic: "+ x +"," + y;
    }
    abstract double aria();
    abstract double perimetrul();
}
```

Toate clasele concrete derivate dintr-o clasa abstracta trebuie sa implementeze metodele abstracte prezente in clasa de baza.

Intr-o aplicatie Java care construiește obiecte grafice pot fi definite cercuri, dreptunghiuri, patrate, etc. Aceste categorii sunt implementate prin intermediul unor clase concrete. Clasa abstracta care traseaza caracteristicile de baza ale obiectelor grafice este reprezentata mai sus: *ObiectGrafic.java*.

✓ `Cerc.java`

```
public class Cerc extends ObiectGrafic {
    protected double raza ;
    public Cerc() {
        super();
        raza = 0;
    }
    public Cerc(double raza0) {
        super();
        raza = raza0;
    }
    public double aria() {
        return java.lang.Math.PI*raza*raza;
    }
    public double perimetrul() {
        return 2*java.lang.Math.PI*raza;
    }
    public String toString() {
        return "Cerc: "+ x +", "+ y + " - "+ raza;
    }
}
```

Subclasele concrete (*Cerc.java* si *Dreptunghi.java*) ofera implementari proprii ale metodelor abstracte, *aria()* si *perimetrul()*, declarate in superclasa *ObiectGrafic.java*.

#### ✓ Dreptunghi.java

```
public class Dreptunghi extends ObiectGrafic {
    protected double lung, lat ;
    public Dreptunghi() {
        super();
        lung = 0;
        lat = 0;
    }
    public Dreptunghi(double lung0, double lat0) {
        super();
        lung = lung0;
        lat = lat0;
    }
}
```

```
    }  
    public double aria() {  
        return lung*lat;  
    }  
    public double perimetrul() {  
        return 2*(lung+lat);  
    }  
    public String toString() {  
        return "Dreptunghi: "+ x +"," + y + " - "+ lung +"," + lat;  
    }  
}
```

Instantierea de obiecte grafice de tip Cerc sau Dreptunghi este realizata in clasa principala a aplicatiei, clasa care contine metoda main(). Metodele toString() sunt utilizate pentru a defini informatii de tip String despre o clasa. Implicit aceste metode afiseaza numele claselor si un cod unic asociat acestora.

#### ✓ ObiecteGrafice.java

```
public class ObiecteGrafice {  
    public static void main(String [] args) {  
        Cerc c = Cerc(2.3);  
        Dreptunghi d = Dreptunghi(4.1, 5.3);  
        System.out.println(c);  
        System.out.println(d);  
        System.out.println(c.aria());  
        System.out.println(d.aria());  
        System.out.println(c.perimetrul());  
        System.out.println(d.perimetrul());  
    }  
}
```

## 6.7 Crearea si utilizarea interfetelor

In limbajul Java, interfetele ofera raspuns lipsei mostenirii multiple. O interfata creaza un *protocol* pe care clasele trebuie sa il implementeze.

Folosind interfetele poate fi precizat pentru o clasa ce sa implementeze, dar nu si cum sa faca acest lucru.

Interfetele reprezinta constructii Java similare claselor abstracte. Ele contin doar definitii de (variabile) constante si metode fara implementare. Dupa definire, o interfata poate fi implementata de un numar nedefinit de clase. Deasemenea, o clasa poate implementa oricate interfete doreste.

Pentru a implementa o interfata, o clasa trebuie sa ofere implementare tuturor metodelor definite in interiorul interfetei respective. Nu prezinta importanta modul in care metodele vor fi implementate in clasa.

Definirea unei interfete se realizeaza in mod similar cu cea a unei clase, prin intermediul cuvintului cheie *interface* :

```
[modificatori] interface nume_interfata [extends lista_super_interfete] {  
    //declaratii metode abstracte  
    //definitii constante  
}
```

Dupa cum se poate observa o interfata poate mosteni una sau mai multe superinterfete.

In declaratia unei interfete poate fi utilizat doar modificatorul public. Toate attributele definite in interiorul unei interfete sunt implicit constante, chiar daca nu sunt declarate folosind modificatorii static si final. Este obligatorie initializarea cu o valoare a acestor attribute de tip constanta. Metodele declarate intr-o interfata sunt implicit abstracte si publice.

Pentru a preciza ca o clasa implementeaza una sau mai multe interfete este utilizata clauza *implements* in declaratia clasei.

```
[modificatori_clasa] class nume_clasa [extends super_clasa][implements lista_interfete] {  
    // corp_clasa  
}
```

In cazul in care se doreste testarea tipului unui obiect poate fi utilizat operatorul *instanceof*. Deasemenea, *instanceof* poate testa si daca o variabila refera un obiect a carei clasa implementeaza o interfata.

Sintaxa generala de utilizare a operatorului instanceof este urmatoarea:

```
referinta_obiect instanceof nume_clasa
```

In cele ce urmeaza consideram clasele *Casa* si *Masina*, si interfata *asigurat*. Interfata defineste metodele abstracte *stabilesteRisc()* si *obțineRisc()*. Deoarece clasele concrete implementeaza interfata, ele sunt obligate sa asigure implementare pentru metodele declarate in aceasta interfata.

✓ asigurat.java

```
public interface asigurat {  
    void stabilesteRisc(String nivel);  
    String obtineRisc();  
}
```

Clasele concrete, definite in aceasta aplicatie, ofera propriile implementari ale metodelor declarate in interfata asigurat. Pe langa aceste metode, in interiorul claselor pot fi intalnite supradefiniri ale metodei toString.

✓ Casa.java

```
public class Casa implements asigurat {  
    String nivelRisc;  
    public Casa() {  
        nivelRisc = " ";  
    }  
    public void stabilesteRisc(String nivel) {  
        if (nivelRisc.equals(" ")) {  
            nivelRisc = nivel;  
        }  
        else {  
            nivelRisc = nivelRisc + ", " + nivel;  
        }  
    }  
    public String obtineRisc() {  
        return nivelRisc;  
    }  
}
```

```
        public String toString() {  
            return "Casa este asigurata pentru: ";  
        }  
    }  
}
```

✓ Masina.java

```
public class Masina implements asigurat {  
    String nivelRisc;  
    public Masina() {  
        nivelRisc = "Accident, incendiu, furt";  
    }  
    public void stabilesteRisc(String nivel) {  
        nivelRisc = nivel;  
    }  
    public String obtineRisc() {  
        return nivelRisc;  
    }  
    public String toString() {  
        return "Masina este asigurata pentru: ";  
    }  
}
```

Clasa principala a aplicatiei include declararea unui tablou de tipul asigurat, denumit *asigurare*, care contine ca si elemente obiecte de tipul Casa si Masina. Dupa instantierea obiectelor, este apelata metoda stabilesteRisc() pentru cateva elemente ale tabloului. Parcurgerea elementelor tabloului este realizata prin intermediul unei iteratii for.

✓ Aplicatie.java

```
public class Aplicatie {  
    public static void main(String [] args) {  
        asigurat[] asigurare = new asigurat[4];  
        asigurare[0] = new Masina();  
        asigurare[1] = new Masina();  
        asigurare[2] = new Casa();  
        asigurare[3] = new Casa();  
        asigurare[0].stabilesteRisc("Accident");  
    }  
}
```

```
    asigurare[2].stabilesteRisc("Inundatie");
    asigurare[2].stabilesteRisc("Incendiu");
    for(int i = 0; i < asigurare.length; i++) {
        System.out.println(asigurare[i].toString() + " " + asigurare[i].obțineRisc());
    }
}
}
```

## 7. Exceptii

In limbajul Java, exceptiile ofera un mecanism eficient de identificare si rezolvare a erorilor. Exceptiile reprezinta situatii care apar in timpul executiei unui program si care determina oprirea acestuia. Java ofera posibilitatea tratarii exceptiilor, prin stabilirea unei cai alternative de continuare a executiei programului.

De exemplu, pot fi generate exceptii in urmatoarele cazuri:

- realizarea unei impartiri la zero (*ArithmeticException*);
- deschiderea unui fisier care nu exista (*FileNotFoundException*);
- accesarea indexului unui tablou, care depaseste limitele acestuia (*IndexOutOfBoundsException*);
- utilizarea unui atribut care nu a fost definit (*NoSuchFieldException*).

### ✓ Exceptii.java

```
public class Exceptii {  
    public static void main(String [] args) {  
        int a = 257;  
        System.out.println("Rezultatul impartirii lui " + a + " la 0 este " + a/0);  
    }  
}
```

Daca este executat programul *Exceptii.java*, se poate observa mesajul de eroare care este generat la realizarea unei impartiri la 0:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Exceptii.main(Exceptii.java:4)
```

In contextul discutiilor despre exceptii, putem defini doua mari categorii: generarea de exceptii folosind instructiunea *throw* si tratarea exceptiilor utilizand constructii de genul *try ... catch*.

### 7.1 Generarea exceptiilor

La aparitia unei exceptii, in limbajul Java, este generat automat un obiect ce contine toata informatia corespunzatoare exceptiei.

Generarea unei exceptii poate fi realizata prin intermediul instructiunii *throw*, astfel:

```
throw obiect_exceptie;
```

Putem considera ca exceptiile reprezinta al doilea tip returnat de catre o metoda. Din acest motiv, o metoda trebuie sa declare tipul exceptiilor care pot fi lansate din interiorul ei. In caz contrar, compilatorul Java genereaza o eroare de compilare.

Declararea tipurilor de exceptii prezente intr-o metoda este realizata folosind cuvantul cheie *throws* in declaratia metodei.

```
[modificatori] tip_metoda nume_metoda( [lista parametrii] ) [throws tip_clasa1, ... ] {
    //corp metoda
}
```

Exceptiile declarate in antetul unei metode extind clasa *Exception*, si cuprind erori care apar datorita problemelor de comunicare cu sisteme externe sau datorita lucrului cu intrarile. Aceste conditii de eroare trebuiesc tratate dinamic, deoarece ele nu pot fi prevenite.

Nu toate tipurile de exceptii trebuiesc declarate folosind clauza *throws*. Astfel de exceptii extind clasa *RuntimeException* si au ca si cauza greselile de programare.

```
void metoda_test() throws IOException {
    //instructiuni
    if (eroare) throw new IOException()
}
```

Clasa *Exceptie* permite testarea valorilor pentru indicii unui tablou *tab*. Daca indexul corespunzator unui element al tabloului depaseste marginile acestuia, atunci este generata (*aruncata*) o exceptie de tipul *IndexOutOfBoundsException*. In cazul in care indexul verificat respecta dimensiunea tabloului, este afisat elementul corespunzator acelu index. In acest caz nu este obligatorie semnarea metodei *main()* cu tipul exceptiei lansate, deoarece exceptia este de tipul eroare logica a programului (*RuntimeException*). In mod normal nu este indicata captarea si tratarea acestor exceptii, ci corectarea lor.

✓ Exceptie.java

```
public class Exceptie {
```

```
public static void main(String [ ] args) throws IndexOutOfBoundsException {  
    int tab [ ] = { 2, 4, 5, 1 };  
    int index = 5;  
    if (index >= tab.length && index <= tab.length)  
        throw new ArrayIndexOutOfBoundsException();  
    else System.out.println("Elementul tab[" + index + "] este " + tab[index]);  
}  
}
```

## 7.2 Categorii de exceptii

Limbajul Java definește o ierarhie de clase pentru tratarea excepțiilor pornind de la clasa *Throwable*, care conține două mari clase: *Error* și *Exception*.

Obiectele de tipul *Error* sunt generate de către mediul de dezvoltare Java și nu pot fi preluate în vederea tratării. De exemplu, dacă se încearcă apelarea unei metode care nu este definită pentru o clasă, va fi generată o eroare de tipul *NoSuchMethodError*.

O subclasă importantă a clasei *Exception* este *RuntimeException*. Această clasă definește excepțiile care pot fi evitate de către programator, printr-o mai bună organizare a surselor – erori logice. La parcurgerea unui tablou poate fi generată o excepție de tipul *RuntimeException (ArrayIndexOutOfBounds)*, dacă indexul tabloului respectiv este depășit. Această eroare poate fi preîntâmpinată dacă utilizatorul ia în calcul limitele tabloului.

Deasemenea, în superclasa *Exception* poate fi întâlnită o altă categorie importantă de excepții, și anume *IOException*. Clasa *IOException* conține excepțiile care apar în contextul lucrului cu intrările/iesirile. Accesarea unui fișier care nu există va determina generarea unei excepții de tipul *IOException, FileNotFoundException*.

### Throwable

#### Exception

- ClassNotFoundException
- CloneNotSupportedException
- IllegalAccessException
- InstantiationException
- InterruptedException
- NoSuchMethodException

## IOException

- EOFException
- FileNotFoundException
- InterruptedIOException
- MalformedURLException
- ProtocolException
- SocketException
- UnknownHostException
- UnkdownServiceException

## RuntimeException

- ArithmeticException
- ArrayStoreException
- ClassCastException
- IllegalArgumentException
  - IllegalThreadStateException
  - NumberFormatException
- IllegalMonitorStateException
- IllegalStateException
- IndexOutOfBoundsException
  - ArrayIndexOutOfBoundsException
  - StringIndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException
- SecurityException

## Error

## LinkageError

- ClassCircularityError
- ClassFormatError
- ExceptionInitializerError
- IncompatibleClassChangeError
  - AbstractMethodError
  - IllegalAccessError
  - InstantiationError
  - NoSuchFieldError
  - NoSuchMethodError
- NoClassDefFoundError
- UnsatisfiedLinkError

```
VerifyError  
ThreadDeath  
VirtualMachineError
```

### 7.3 Tratarea exceptiilor folosind try ... catch ... finally

Java ofera o solutie eficienta de rezolvare a erorilor care apar intr-un program prin intermediul mecanismului de tratare a exceptiilor. Implementarea unei astfel de solutii se face folosind constructii de genul *try ... catch ... finally*.

Sintaxa generala pentru o instructiune *try ... catch ... finally* este urmatoarea:

```
try {  
    //instructiuni supravegheate  
}  
catch(Exceptie1 e1) {  
    //tratare exceptie e1 de tip Exceptie1  
}  
catch(Exceptie2 e2) {  
    //tratare exceptie e2 de tip Exceptie2  
}  
finally {  
    //instructiuni care se executa neconditionat  
}
```

Blocul *try* este utilizat pentru a delimita instructiunile ce vor fi urmarile in vederea identificarii exceptiilor. Daca este lansata o eroare, se suspenda executia restului de instructiuni din blocul *try*, si sunt verificate blocurile *catch*. Eroarea este tratata de primul bloc *catch* care contine tipul erorii sau un supertip al acestei erori.

In cazul in care eroarea nu este tratata de niciun bloc *catch*, ea va fi returnata in metoda apelanta. Netratarea erorii, si returnarea ei pana in varful ierarhiei, va determina in cele din urma oprirea programului si afisarea unui mesaj de eroare.

✓ TratareExceptii.java

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

```
        try {
            nr = Integer.parseInt( args[0] );
        }
        catch(NumberFormatException e) {
            System.out.println("NumberFormatException");
            nr = 1;
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException");
            nr = 2;
        }
        finally {
            nr++;
        }
        System.out.println("nr= " + nr);
    }
}
```

In clasa *TrateazaExceptii*, poate fi observata o modalitate de tratare a erorilor care sunt generate ca urmare a validarii valorii atribuite elementului corespunzator indexului 0 al listei de parametrii, *args[]*. Daca este omisa introducerea unei valori pentru parametrul *args[0]* in linia de comanda, se va genera o exceptie de tipul *ArrayIndexOutOfBoundsException*. Daca valoarea atribuita parametrului *args[0]* nu poate fi convertita la tipul intreg, se genereaza o exceptie de tipul *NumberFormatException*. Se poate observa ca instructiunea din interiorul blocului *finally* este executata intotdeauna.

## 7.4 Definirea de exceptii utilizator

Limbajul Java ofera posibilitatea definirii propriilor exceptii, pentru a pune in evidenta erori care nu au fost prevazute in ierarhia exceptiilor standard. O clasa de exceptii nou creata trebuie sa extinda unul din tipurile existente de exceptii. In general, extinderea se realizeaza dintr-o clasa a carei semnificatie este asemanatoare clasei *Exception*.

✓ Aplicatie.java

```
import java.io.*;
```

```
public class Aplicatie {
public static void main(String [ ] args) {
    int temperatura;
    BufferedReader fluxIn = new BufferedReader(
        new InputStreamReader(System.in));
    String linie;
    try {
        System.out.print("Temperatura: ");
        linie = fluxIn.readLine();
        temperatura = Integer.parseInt(linie);
    }
    catch(NumberFormatException e){
        System.out.println("Temperatura cafea este de tip intreg.");
        return;
    }
    catch(IOException e) {
        return;
    }
    CeascaCafea ceasca = new CeascaCafea();
    ceasca.stabilesteTemperatura(temperatura);
    Client client = new Client();
    Cafea.servesteCafea(client, ceasca);
}
}
```

✓ Cafea.java

```
public class Cafea {
    public static void servesteCafea(Client client, CeascaCafea ceasca) {
        try {
            client.beaCafea(ceasca);
            System.out.println("Cafeaua este buna.");
        }
        catch(PreaRece e) {
            System.out.println("Cafeaua este prea rece.");
        }
        catch(PreaCalda e) {
            System.out.println("Cafeaua este prea calda.");
        }
    }
}
```

```
    }  
  }  
}
```

✓ CeascaCafea.java

```
public class CeascaCafea {  
    private int temperatura;  
    public void stabilesteTemperatura(int temperatura) {  
        this.temperatura = temperatura;  
    }  
    public int determinaTemperatura() {  
        return temperatura;  
    }  
}
```

✓ Client.java

```
public class Client {  
    private static final int preaRece = 65;  
    private static final int preaCalda = 85;  
    public void beaCafea(CeascaCafea ceasca) throws PreaRece, PreaCalda {  
        int temperatura = ceasca.determinaTemperatura();  
        if (temperatura <= preaRece) {  
            throw new PreaRece(temperatura);  
        }  
        else if (temperatura >= preaCalda) {  
            throw new PreaCalda(temperatura);  
        }  
    }  
}
```

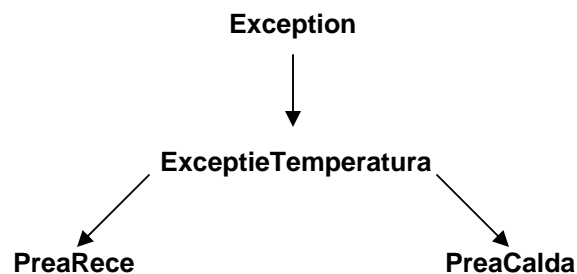
✓ Exceptii.java

```
abstract class ExceptieTemperatura extends Exception {  
    private int temperatura;  
    public ExceptieTemperatura(int temperatura) {  
        this.temperatura = temperatura;  
    }  
}
```

```
        public int determinaTemperatura() {  
            return temperatura;  
        }  
    }  
}  
  
class PreaRece extends ExceptieTemperatura {  
    public PreaRece(int temperatura) {  
        super(temperatura);  
    }  
}  
  
class PreaCalda extends ExceptieTemperatura {  
    public PreaCalda(int temperatura) {  
        super(temperatura);  
    }  
}
```

Aplicatia de mai sus cuprinde clasele: *Cafea*, *CeascaCafea*, *Client*, *Aplicatie*, *ExceptieTemperatura*, *PreaRece*, *PreaCalda*, necesare pentru a gestiona situatiile care apar intr-o cafenea, in ceea ce priveste temperatura optima de servire a unei cesti de cafea.

Cazurile exceptionale sunt definite prin intermediul clasei abstracte *ExceptieTemperatura*, care extinde clasa *Exception*, si a claselor *PreaCalda* si *PreaRece*, care sunt subtipuri ale superclasei *ExceptieTemperatura*.



Exceptiile de tipul *PreaRece*, *PreaCalda* pot fi lansate in metoda *beaCafea()* din interiorul clasei *Client*. Tratarea exceptiilor generate se realizeaza folosind o constructie *try ... catch*, in metoda *servesteCafea* din clasa *Cafea*.

## 8. Operatii de intrare/iesire

### 8.1 Definirea conceptului de flux de date

Un flux de date (*stream*) reprezinta un canal de comunicatie prin care datele circula de la o sursa catre o destinatie. Un flux care citeste date de la o sursa poarta denumirea de flux de intrare (*input stream*), iar un flux care scrie date la o destinatie se numeste flux de iesire (*output stream*).

In Java, fluxurile si operatiile corespunzatoare sunt implementate prin intermediul claselor din pachetul *java.io*. Deci orice program care utilizeaza operatii de intrare/iesire trebuie sa contina instructiunea *import* pentru pachetul *java.io*: *import java.io.\**.

In general, pentru lucrul cu fluxuri de date sunt necesari urmasorii pasi: deschidere flux, citire/scriere informatie, inchidere flux dupa utilizare. Este recomandata inchiderea fluxurilor de date neutilizate, pentru a elibera resursele ocupate de catre acestea. Pentru a inchide un flux de date, in limbajul Java, este utilizata metoda *close()*.

Din punctul de vedere al tipului de date pe care opereaza, fluxurile pot fi clasificate in fluxuri de octeti (*mod binar*) si fluxuri de caractere (*mod text*).

### 8.2 Fluxuri standard de intrare/iesire

Fluxurile de intrare/iesire standard exista implicit in orice aplicatie Java, si apar sub forma de campuri statice ale clasei *System*:

- flux de intrare: *public static final InputStream in*
- flux de iesire: *public static final PrintStream out*

Pentru citirea datelor de la tastatura pot fi utilizate metode ale clasei *InputStream*:

- *int read()* - citeste un caracter si returneaza ca intreg;
- *int read(char c[])* - citeste un sir de caractere de lungime egala cu tabloul *c* si il completeaza in *c*;
- *int read(byte b[])* - citeste un sir de octeti de lungime egala cu tabloul *b* si il completeaza in *b*;

Clasa Intraire permite citirea unui caracter de la tastatura si afisarea codului ASCII corespunzator.

✓ Intraire.java

```
import java.io.*;
public class Intraire {
    public static void main(String args[ ]) throws IOException {
        char c;
        System.out.print("Introduceti un caracter ");
        c=(char)System.in.read();
        System.out.println("[ "+c+"");
        System.out.println("[ "+(int)c+"");
    }
}
```

Afisrea datelor se face folosind una din metodele de scriere:

- int write() - scrie un caracter la iesire;
- int write(char c[]) - scrie un tablou de caractere la iesire;
- int println(String s) - scrie un sir de caractere la iesire;

Fluxurile pot fi clasificate dupa cum urmeaza:

- fluxuri pentru citire/scriere:
  1. fluxuri de caractere: SringerReader, StringWriter, FileReader, FileWriter, CharArrayReader, CharArrayWriter;
  2. fluxuri de octeti: ByteArrayReader, ByteArrayWriter, FileInputStream, FileOutputStream;
- fluxuri pentru procesarea datelor:
  1. fluxuri de caractere: BufferedReader, BufferedWriter, InputStreamReader, OutputStreamWriter, PrintWriter;
  2. fluxuri de octeti: PrintStream, BufferedInputStream, BufferedOutputStream, DataInputStream, DataOutputStream;

### 8.3 Utilizarea fluxurilor de date

✓ Web.java

```
import java.io.*;
import java.net.*;
class Web {
    public static void main(String args[ ]) throws IOException {
        String buffer;
        BufferedReader in = new BufferedReader(
            new InputStreamReader((new URL("http://www.csid.upt.ro")).openStream()));
        while ((buffer = in.readLine()) != null) {
            System.out.println(buffer);
        }
        in.close();
    }
}
```

✓ File.java

```
import java.io.*;
class File {
    public static void main(String args[ ]) throws IOException {
        String buffer;
        BufferedReader in = new BufferedReader(
            new InputStreamReader(new FileInputStream("file.txt" )));
        while ((buffer = in.readLine()) != null) {
            System.out.println(buffer);
        }
        in.close();
    }
}
```

✓ ReadToFile.java

```
import java.io.*;
class ReadToFile {
    public static void main(String[ ] args) throws IOException {
```

```
String buffer;

BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
PrintWriter file = new PrintWriter(new FileWriter("file.txt"));
buffer = in.readLine();
while(buffer.charAt(0) != '.'){
    file.println(buffer);
    buffer = in.readLine();

}
in.close();
file.close();
}
}
```