Curs 4

Programare Orientată pe Obiecte în limbajul Java

Programare Orientată pe Obiecte

Cuprins

- Exemplu clasa Complex
- Variabie şi metode de instanță/clasă
- Blocuri statice
- Clasa Object
- Polimorfism
- Tipul enumerare
- Clase imbricate
- Clase şi metode abstracte

- Excepții

Exemplu

```
public class Complex {
      private double a, b;
      ...
      public Complex aduna(Complex comp) {
            return new Complex(a + comp.a, b + comp.b);
      }
      public boolean equals(Object obj) {
            if (obj == null) return false;
            if (!(obj instanceof Complex)) return false;
            Complex comp = (Complex) obj;
            return ( comp.a==a && comp.b==b);
      }
      public String toString() {
            if (b > 0) return a + ”+” + b + ”*i”;
             return a + “” + b + ” *i";
      }
}
...
Complex c1 = new Complex(1,2);
Complex c2 = new Complex(2,3);
System.out.println(c1.aduna(c2));              // 3.0 + 5.0i
System.out.println(c1.equals(c2));              // false
```

Variabile de instanță şi variabile de clasă

```
class Exemplu {
    int x ; //variabila de instanta
}
```

- variabilă de instanță: la fiecare creare a unui obiect al clasei Exemplu sistemul alocă o zonă de memorie separată pentru memorarea valorii lui x.

```
class Exemplu {
    static int sx ; //variabila de clasă
}
```

- Pentru variabilele de clasă (statice) sistemul alocă o singură zonă de memorie la care au acces toate instanțele clasei respective, ceea ce înseamnă că dacă un obiect modifică valoarea unei variabile statice ea se va modifica şi pentru toate celelalte obiecte.

Variabile de clasă

- Deoarece nu depind de o anumită instanță a unei clase, variabilele statice pot fi referite şi sub forma:

  NumeClasa.numeVariabilaStatica

  Ex. Exemplu.sx

- Inițializarea variabilelor de clasă se face o singură dată, la încărcarea în memorie a clasei respective

```
class Exemplu {
    static final double PI = 3.14;
    static long nrInstante = 0;
    static Point p = new Point(0,0);
}
```

Variabile de instanță şi variabile de clasă

```
class Exemplu {
    int x ; // Variabila de instanta
    static long n; // Variabila de clasa
}
    ...
    Exemplu o1 = new Exemplu();
    Exemplu o2 = new Exemplu();
    o1.x = 100;
    o2.x = 200;
    System.out.println(o1.x); // Afiseaza 100
    System.out.println(o2.x); // Afiseaza 200
    o1.n = 100;
    System.out.println(o2.n); // Afiseaza 100
    o2.n = 200;
    System.out.println(o1.n); // Afiseaza 200
    System.out.println(Exemplu.n); // Afiseaza 200
    // o1.n, o2.n si Exemplu.n sunt referinte la aceeasi
        // valoare
```

Metode de instanță şi metode de clasă

- **metodele de instanță operează atât pe variabilele de instanță cât şi pe cele statice ale clasei;**
- **metodele de clasă operează doar pe variabilele statice ale clasei.**

```
class Exemplu {
    int x ;      // Variabilă de instanță
    static long n; // Variabilă de clasă
    void metodaDeInstanta() {
        n ++; // Corect
        x --; // Corect
    }
    static void metodaStatica() {
        n ++; // Corect
        x --; // Eroare la compilare !
    }
}
```

Metode de instanță şi metode de clasă

- Intocmai ca şi la variabilele statice, întrucât metodele de clasă nu depind de starea obiectelor clasei respective, apelul lor se poate face şi sub forma:

  NumeClasa.numeMetodaStatica

```
Exemplu.metodaStatica(); // Corect
Exemplu obj = new Exemplu();
obj.metodaStatica(); // Corect
```

- Metodele de instanță nu pot fi apelate decât pentru un obiect al clasei respective:

```
Exemplu.metodaDeInstanta(); // Eroare
Exemplu obj = new Exemplu();
obj.metodaDeInstanta(); // Corect
```

Utilitatea membrilor de clasă

- **folosiți pentru a pune la dispoziție valori şi metode independente de starea obiectelor dintr-o anumită clasă.**

- **Declararea eficientă a constantelor**

```
class Exemplu {
    static final double PI = 3.14;
    // Variabila finala de clasa
}
```

- **Numărarea obiectelor unei clase**

```
class Exemplu {
    static long nrInstante = 0;
    Exemplu() {
        // Constructorul este apelat la fiecare instantiere
        nrInstante ++;
    }
}
```

- **Implementarea funcțiilor globale**

Blocuri statice de inițializare

```
static {
    // Bloc static de initializare;
    ...
}
public class Test {
    // Declaratii de variabile statice
    static int x = 0, y, z;
    // Bloc static de initializare
    static {
        System.out.println("Initializam...");
        int t=1;
        y = 2;
        z = x + y + t;
    }
    Test() { ... }
    }
}
```

Blocuri statice de inițializare

- Variabilele statice ale unei clase sunt inițializate la un moment care precede prima utilizare activă a clasei respective.
- Momentul efectiv depinde de implementarea maşinii virtuale Java şi poartă numele de inițializarea clasei. În această etapă sunt executate şi blocurile statice de inițializare ale clasei.
- Variabilele referite într-un bloc static de inițializare trebuie să fie obligatoriu de clasă sau locale blocului.

Clasa Object

Object este superclasa tuturor claselor.

```
class Exemplu {}
class Exemplu extends Object {}
```

- clone
- equals
- finalize
- toString.

```
Exemplu obj = new Exemplu();
System.out.println("Obiect=" + obj);
//echivalent cu
System.out.println("Obiect=" + obj.toString());
```

Polimorfism (1)

- **Supraîncărcarea (overloading)**
- **Supradefinirea (overriding)**

```
class A {
    void metoda() {
        System.out.println("A: metoda fara parametru");
    }
    // Supraîncărcare
    void metoda(int arg) {
        System.out.println("A: metoda cu un
          parametru");
    }
}
class B extends A {
    // Supradefinire
    void metoda() {
        System.out.println("B: metoda fara parametru");
    }
}
```

Polimorfism (2)

O metodă supradefinită poate :

• să ignore codul metodei părinte:

```
B b = new B();
b.metoda();
// Afişează "B: metoda fara parametru"
```

• să extindă codul metodei părinte:

```
class B extends A {
// Supradefinire prin extensie
    void metoda() {
        super.metoda();
        System.out.println("B: metoda fara parametru");
    }
}
. . .
B b = new B();
b.metoda();
/* Afişează ambele mesaje:
"A: metoda fara parametru"
"B: metoda fara parametru" */
```

- **În Java nu este posibilă supraîncărcarea operatorilor.**

Tip - Subtip

- **int metoda() { return 1.2;} // Eroare**
- **int metoda() { return (int)1.2;} // Corect**
- **double metoda() {return (float)1;} // Corect**

Clasă – Subclasă

class Patrat extends Poligon { … }

```
Poligon metoda1( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p; // Corect
        else
                return t; // Corect
}
Patrat metoda2( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p; // Eroare
        else
                return t; // Corect
}
```

Legare statică/dinamică – static/dynamic binding

```
class Vehicle {
    public void drive() {
        System.out.println("A");
    }
}

class Car extends Vehicle {
    public void drive() {
        System.out.println("B");
    }
}
class TestCar {
    public static void main(String args[]) {
        Vehicle v;
        Car c;
        v = new Vehicle();
        c = new Car();
        v.drive();
        c.drive();
        v = c;
        v.drive();
    }
}
```

Tipuri de date enumerare

```
enum

public class CuloriSemafor {
      public static final int ROSU = -1;
      public static final int GALBEN = 0;
      public static final int VERDE = 1;
}

...
// Exemplu de utilizare
if (semafor.culoare == CuloriSemafor.ROSU)
semafor.culoare = CuloriSemafor.GALBEN;

// Doar de la versiunea 1.5 !

public enum CuloriSemafor { ROSU, GALBEN, VERDE };
// Utilizarea structurii se face la fel
```

Clase imbricate

- o clasă membră a unei alte clase, numită şi clasă de acoperire.

```
class ClasaDeAcoperire{
    class ClasaImbricata1 {
        // Clasa membru
        // Acces la membrii clasei de acoperire
    }
    void metoda() {
        class ClasaImbricata2 {
            // Clasa locala metodei
            // Acces la mebrii clasei de acoperire si
            // la variabilele finale ale metodei
        }
    }
}
```

- **Identificare claselor imbricate**
  - **ClasaDeAcoperire.class**
  - **ClasaDeAcoperire$ClasaImbricata1.class**
  - **ClasaDeAcoperire$ClasaImbricata2.class**

Clase imbricate

```
class ClasaDeAcoperire{
    private int x=1;
    class ClasaImbricata1 {
        int a=x;
    }
    void metoda() {
        final int y=2;
        int z=3;
        class ClasaImbricata2 {
          int b=x;
          int c=y;
          int d=z; // Incorect
        }
    }
}
```

Clase imbricate

- O clasă imbricată membră (care nu este locală unei metode) poate fi referită din exteriorul clasei de acoperire folosind expresia

  ClasaDeAcoperire.ClasaImbricata
- Clasele membru pot fi declarate cu modificatorii public, protected, private sau implicit pentru a controla nivelul lor de acces din exterior.
- Clasa care conține alte clase poate avea doar modificatorul public şi cel implicit!
- Pentru clasele imbricate locale unei metode nu sunt permişi aceşti modificatori.
- Toate clasele imbricate pot fi declarate folosind modificatorii abstract şi final.

Clase anonime

- clase imbricate locale, fără nume, utilizate doar pentru instanțierea unui obiect de un anumit tip.
- sunt foarte utile în crearea unor obiecte ce implementează o anumită interfață sau extind o anumită clasă abstractă.

Clase şi metode abstracte

```
[public] abstract class ClasaAbstracta ... {
    // Declaratii uzuale
    // Declaratii de metode abstracte
}
```

Metode abstracte

```
abstract class ClasaAbstracta {
    abstract void metodaAbstracta(); // Corect
    void metoda(); // Eroare
}
```

- O clasă abstractă poate să nu aibă nici o metodă abstractă.
- O metodă abstractă nu poate apărea decât într-o clasă abstractă.
- Orice clasă care are o metodă abstractă trebuie declarată ca fiind abstractă.

Exemple:

Number: Integer, Double, ...

Component: Button, List, ...

Excepții

- Ce sunt excepțiile
- ”Prinderea” şi tratarea excepțiilor
- ”Aruncarea” excepțiilor
- Avantajele tratării excepțiilor
- Ierarhia claselor ce descriu excepții
- Excepții la execuție
- Crearea propriilor excepții

Ce sunt excepțiile ?

Excepție = ”eveniment excepțional”

```
public class Exemplu {
   public static void main(String args[]) {
     int v[] = new int[10];
     v [10] = 0; //Exceptie !
     System.out.println("Aici nu se mai ajunge..");
   }
}
```

"Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException :10
at excepții.main (excepții.java:4)“

- ”throw an exception”
- ”exception handler”
- ”catch the exception”

Tratarea erorilor nu mai este o opțiune ci o constrângere!

”Prinderea” şi tratarea excepțiilor

try - catch - finally

```
try {
    … // Instrucțiuni care pot genera excepții
}
catch (TipExceptie1 variabila) {
    … // Tratarea excepțiilor de tipul 1
}
catch (TipExceptie2 variabila) {
    … // Tratarea excepțiilor de tipul 2
}
. . .
finally {
    … // Cod care se execută indiferent
    … // dacă apar sau nu excepții
}
```

Citirea unui fişier (1)

```
public static void citesteFisier(String fis) {
      FileReader f = null;
      // Deschidem fisierul
      f = new FileReader(fis);
      // Citim si afisam fisierul caracter cu
      //  caracter
      int c;
      while ( (c=f.read()) != -1)
            System.out.print((char)c);
      // Inchidem fisierul
      f.close();
}
```

Pot provoca excepții:

- Constructorul lui FileReader
- read
- close

Citirea unui fişier (2)

```
public static void citesteFisier(String fis) {
      FileReader f = null;
      try {
                  // Deschidem fisierul
            f = new FileReader(fis);
                  // Citim si afisam fisierul caracter cu caracter
            int c;
            while ( (c=f.read()) != -1)
              System.out.print((char)c);
      }
      catch (FileNotFoundException e) {
            //Tratam un tip de exceptie
         System.err.println("Fisierul nu a fost gasit");
      }
      catch (IOException e) {
            //Tratam alt tip de exceptie
         System.out.println("Eroare la citire");
         e.printStackTrace();
      }
```

Citirea unui fişier (3)

```
        finally {
           if (f != null) {
            // Inchidem fisierul
              try {
                f.close();
              }
              catch (IOException e) {
                  System.err.println("Fisierul nu poate fi
           inchis!");
                  e.printStackTrace(); }
            } // if
        } //finally
}
```

”Aruncarea” excepțiilor (1)

- A doua metodă de lucru cu excepțiile
- Se utilizează clauza throws în antetul metodelor care pot genera excepții:

```
[modific] TipReturnat metoda([argumente])
    throws TipExceptie1, TipExceptie2, ...
{
...
}
```

Atentie !!!

- **O metoda care nu tratează o anumita exceptie trebuie obligatoriu să o ”arunce”.**

”Aruncarea” excepțiilor (2)

```
public class CitireFisier {
   public static void citesteFisier(String fis) throws
    FileNotFoundException, IOException
    {
       FileReader f = null;
       f = new FileReader(fis);
       int c;
       while ( (c=f.read()) != -1)
           System.out.print((char)c);
       f.close();
    }
```

”Aruncarea” excepțiilor (3)

```
  public static void main(String args[]) {
     if (args.length > 0) {
          try {
               citesteFisier(args[0]);
          }
          catch (FileNotFoundException e){
               System.err.println("Fisierul n-a fost gasit");
           }
          catch (IOException e) {
               System.out.println("Eroare la citire");
          }
     }
     else
          System.out.println("Lipseste numele fisierului");
  } // main
} // clasa
```

try - finally

```
public static void citesteFisier (String fis) throws
    FileNotFoundException, IOException
{
     FileReader f = null;
     try {
          f = new FileReader (fis);
          int c;
          while ( (c=f.read()) != -1)
               System.out.print((char)c);
     }
     finally {
          if (f!=null)
               f.close();
     }
}
public static void main (String args[]) throws
    FileNotFoundException, IOException {
                    citesteFisier(args[0]);
}
```

Instrucțiunea throw

- Aruncarea explicită de excepții:

Exemplu:

```
throw new IOException("Exceptie I/O");
```

Sau:

```
if (index >= vector.length)
    throw new
       ArrayIndexOutOfBoundsException();
```

Sau:

```
catch(Exception e) {
    System.out.println ("A aparut o exceptie);
    throw e;
}
```

Avantajele tratării excepțiilor

1. Separarea codului
2. Propagarea erorilor
3. Gruparea erorilor după tip.

Separarea codului (1)

```
citesteFisier {
    deschide fişierul;
    determină dimensiunea fişierului;
    alocă memorie;
    citeşte fişierul în memorie;
    închide fişierul;
}
```

Separarea codului (2)

Cod ”tradițional” (”spaghetti”):

```
int citesteFisier() {
    int codEroare = 0;
    deschide fisierul;
    if (fisierul s-a deschis) {
        determina dimensiunea fisierului;
        if (s-a determinat dimensiunea) {
            aloca memorie;
            if (s-a alocat memorie) {
                citeste fisierul in memorie;
                if (nu se poate citi din fisier) {
                    codEroare = -1;
                }
        } else { ...
}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
return codEroare; }
```

Separarea codului (3)

```
int citesteFisier() {
        try {
                deschide fişierul;
                determină dimensiunea fişierului;
                alocă memorie;
                citeşte fişierul în memorie;
                închide fişierul;
        }
        catch (fişierul nu s-a deschis)
                {tratează eroarea;}
        catch (nu s-a determinat dimensiunea)
                {tratează eroarea;}
        catch (nu s-a alocat memorie)
                {tratează eroarea}
        catch (nu se poate citi din fişier)
                {tratează eroarea;}
        catch (nu se poate închide fişierul)
                {tratează eroarea;}
}
```

Propagarea erorilor

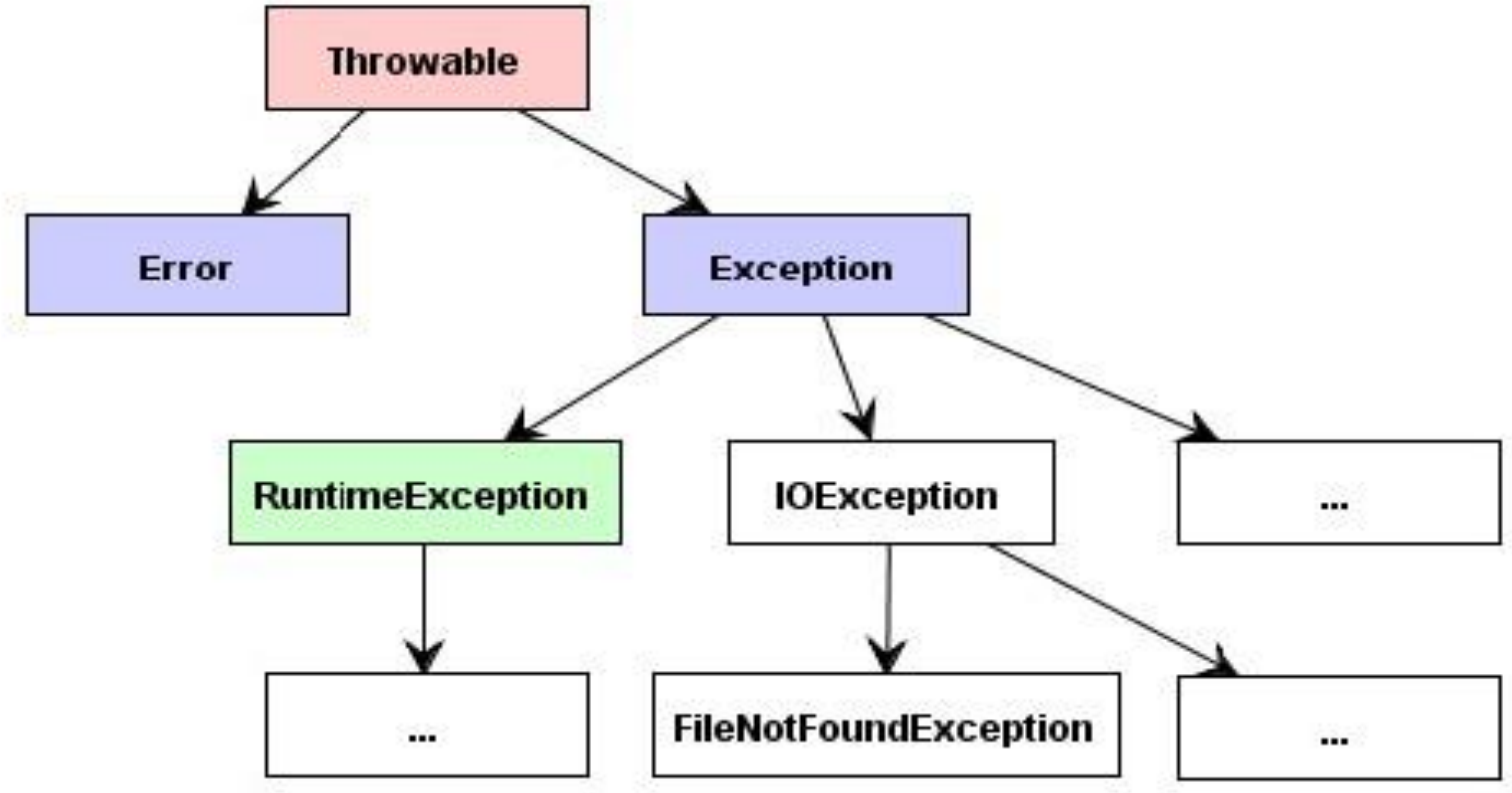
```
int metoda1() {
	try {
		metoda2();
	}
	catch (TipExceptie e) {
		//proceseazaEroare;
	}
	...
}
int metoda2() throws TipExceptie {
	metoda3();
	...
}
int metoda3() throws TipExceptie {
	citesteFisier();
	...
}
```

Gruparea erorilor după tipul lor

- Fiecare tip de excepție este descris de o clasă.
- Clasele sunt organizate ierarhic.

```
try {
      FileReader f = new FileReader("input.dat");
      // Exceptie posibilă: FileNotFoundException
}
catch (FileNotFoundException e) {
      // Exceptie specifică provocată de absența
      // fişierului ’input.dat’
} // sau
catch (IOException e) {
      // Exceptie generică provocată de o operatie IO
} // sau
catch (Exception e) {
      // Cea mai generică excepție soft
} //sau
catch (Throwable e) {
      // Superclasa excepțiilor
}
```

Ierarhia claselor ce descriu excepții



Metode:

- getMessage
- printStackTrace
- toString

Excepții la execuție

RuntimeException

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException

…

```
int v[] = new int[10];
try {
      v[10] = 0;
} catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Atentie la indecsi!");
      e.printStackTrace();
} // Corect, programul continuă

v[11] = 0;
/* Nu apare eroare la compilare dar apare exceptie la
    executie si programul va fi terminat.*/
System.out.println("Aici nu se mai ajunge...");
```

ArithmeticException

- Împărțirea la 0 va genera o excepție doar dacă tipul numerelor împărțite este aritmetic întreg.
- În cazul tipurilor reale (float şi double) nu va fi generată nici o excepție, ci va fi furnizat ca rezultat o constantă care poate fi, funcție de operație, Infinity, -Infinity, sau Nan.

```
int a=1, int b=0;
System.out.println(a/b); // Exceptie la executie!
double x=1, y=-1, z=0;
System.out.println(x/z); // Infinity
System.out.println(y/z); // -Infinity
System.out.println(z/z); // NaN
```

Crearea propriilor excepții (1)

```
public class ExceptieProprie extends
    Exception {
     public ExceptieProprie(String mesaj) {
          super(mesaj);
          /* Apeleaza constructorul superclasei
            Exception */
     }
}
```

Exemplu:

```
class ExceptieStiva extends Exception {
     public ExceptieStiva(String mesaj) {
          super(mesaj);
     }
}
```

Crearea propriilor excepții (2)

```
class Stiva {
    int elemente[] = new int[100];
    int n=0; //numarul de elemente din stiva
    public void adauga(int x) throws ExceptieStiva {
        if (n==100)
            throw new ExceptieStiva("Stiva este plina!");
        elemente[n++] = x;
    }
    public int scoate() throws ExceptieStiva {
        if (n==0)
            throw new ExceptieStiva("Stiva este
          goala!");
    return elemente[--n];
    }
}
```