Curs 4

Programare Orientată pe Obiecte în limbajul Java

Programare Orientată pe Obiecte

Cuprins

- Tipul enumerare
- Clase imbricate
- Clase şi metode abstracte

- Excepții

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 pentru a controla nivelul lor de acces din exterior.
- 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(numeFisier);
                int c;
                while ( (c=f.read()) != -1)
                        System.out.print((char)c);
        }
        finally {
                if (f!=null)
                        f.close();
        }
}
```

- Aruncarea erorilor din metoda main:

```
public static void main(String args[])
throws FileNotFoundException, IOException {
    citeste(args[0]);
}
```

Instrucțiunea throw

- Aruncarea explicită de excepții

```
throw new IOException("Exceptie I/O");
...
if (index >= vector.length)
    throw new ArrayIndexOutOfBoundsException();
...
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

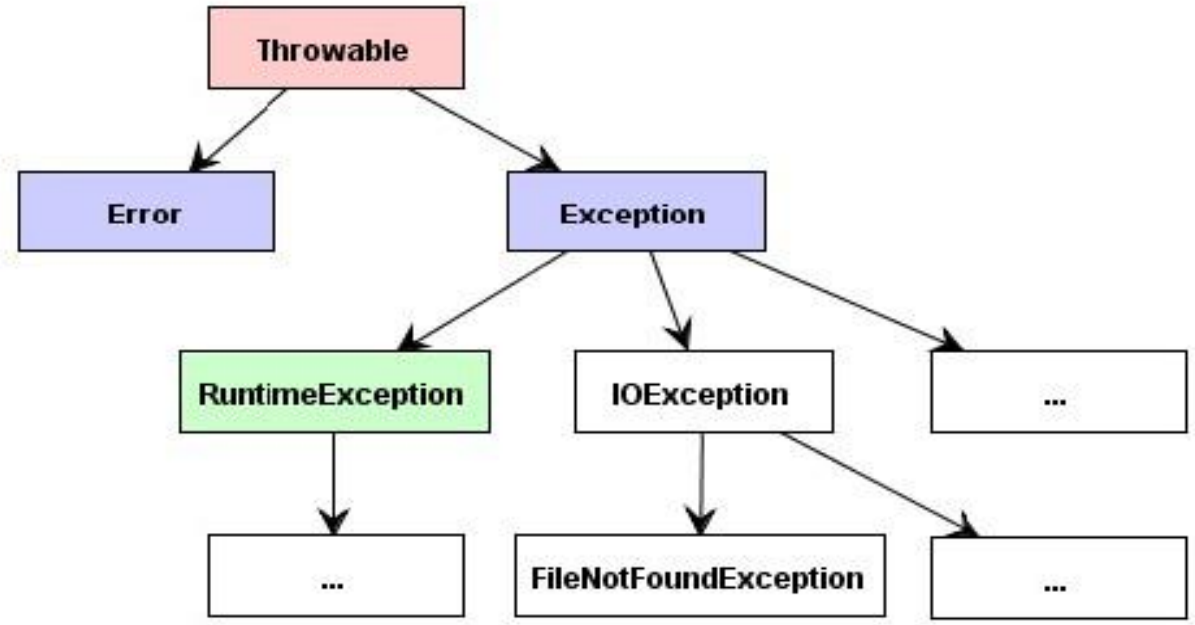
```
int metoda1() {
	try {
		metoda2();
	}
	catch (TipExceptie e) {
		//proceseazaEroare;
	}
	...
}
int metoda2() throws TipExceptie {
	metoda3();
	...
}
int metoda3() throws TipExceptie {
	citesteFisier();
	...
}
```

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];
    }
}
```