Clase incluse

Programare Orientată pe Obiecte

Clase incluse (interne, imbricate, *nested classes*)

- **Clase declarate în interiorul unei alte clase**

- Reprezintă o funcționalitate importantă
  - permit gruparea claselor care sunt legate logic
  - controlul vizibilității uneia din cadrul celorlalte.

- Se comportă ca un **membru** al clasei => o clasă inclusa are acces la toți membrii clasei de care aparține (*outer class*), inclusiv cei private!

- Mai multe tipuri, în funcție de modul de a le instanția şi de relația lor cu clasa exterioră:
  - clase **interne** normale (*regular inner classes*) - **membru**
  - clase **interne metodelor** (*method-local inner classes*) sau blocurilor - **locale**
  - clase **anonime** (*anonymous inner classes*)
  - clase incluse statice (*static nested classes*)

Clase interne

- 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 membrii clasei de
          acoperire si la variabilele
          finale ale metodei */
      }
   }
}
```

Clase interne

- **Identificare claselor imbricate**

  ClasaDeAcoperire.class

  ClasaDeAcoperire$ClasaImbricata1.class

  ClasaDeAcoperire$ClasaImbricata2.class

Clase interne

```
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
      }
  }
}
```

Modificatori de acces

- Clasele membru (1) pot fi declarate cu modificatorii **public, protected, private** sau **implicit** pentru a controla nivelul lor de acces din exterior.

- Pentru clasele imbricate locale unei metode(2) nu sunt permişi aceşti modificatori!

- Toate clasele imbricate pot fi declarate folosind modificatorii **abstract** şi **final**.

- **Clasa care conţine alte clase poate avea doar modificatorul public şi cel implicit!**

Clase interne membru (1)

- Compilatorul creează fişiere .class separate pentru fiecare clasă internă
- Clasa internă poate fi referită din exteriorul clasei de acoperire folosind expresia

  `ClasaAcoperire.ClasaInternă`
- nu este permisă execuția fişierului

  `ClasaAcoperire$ClasaInternă.class`
- Dintr-o clasă internă putem accesa referința la clasa de acoperire (externa):

  `ClasăAcoperire.this`
- Două modalități de a obține o instanță a clasei interne:

1. definim o metodă (getInnerInstance) care creează şi întoarce o astfel de instanță;
2. instanțiem efectiv clasa internă;

   Pentru a instanția clasa internă avem nevoie de o instanta a clasei externe

Clase interne membru (1) - Exemplu

```
class Outer {
   class Inner {
        private int i;
        public Inner (int i) {
              this.i = i;
        }
        public int value () { return i; }
   }
   public Inner getInnerInstance () {
        Inner in = new Inner (11);
        return in;
   }
}
public class Test {
   public static void main(String[] args) {
        Outer out = new Outer ();
        Outer.Inner in1 = out.getInnerInstance();
        Outer.Inner in2 = out.new Inner(10);
        System.out.println(in1.value());
        System.out.println(in2.value());
   }
}
```

Problemă

- Ce se petrece dacă declarăm clasa Inner cu modificatorul private?

```
class Outer {
   private class HiddenInner {
        private int i;
        public HiddenInner (int i) {
               this.i = i; }
        public int value () {
               return i;
        }
   }
   public HiddenInner getInnerInstance () {
        HiddenInner in = new HiddenInner (11);
        return in;
   }
}
```

- În acest mod, vizibilitatea ei a fost redusă pentru că nu poate fi instanțiată decât în clasa Outer!

OBS

- O putem accesa din exteriorul clasei Outer?

```
Outer.HiddenInner in1 =
  out.getInnerInstance();
Outer.HiddenInner in2 = new Outer().new
  HiddenInner(10);
```

- Observatii

```
Outer out=new Outer();
Object in1 = out.getInnerInstance();
System.out.println(in1.value()); //NU
System.out.println ((Outer.HiddenInner)
  in1.value()); //NU merge
```

Clase interne membru (1) – Exemplu clasă ascunsă

```
interface Hidden {
   public int value();
}
class Outer {
   private class HiddenInner implements Hidden {
       private int i;
       public HiddenInner (int i) {
              this.i = i;
       }
       public int value () {
              return i;
       }
   }
   public Hidden getInnerInstance () {
       HiddenInner in = new HiddenInner(11);
       return in;
   }
}
```

Clase interne membru (1) – Exemplu clasă ascunsă

```
public class Test {

  public static void main(String[] args) {

    Outer out = new Outer();

    Outer.HiddenInner in1=out.getInnerInstance();
  //eroare, tipul Outer.HiddenInner nu e vizibil!

    Outer.HiddenInner in2 = new Outer().new
            HiddenInner(10); // din nou eroare

    Hidden in3=out.getInnerInstance();
  // acces corect la o instanta HiddenInner

    System.out.println(in3.value());
  }
}
```

Clase interne în metode - locale

```
interface Hidden {
   public int value ();
}
class Outer {
   public Hidden getInnerInstance() {
        class FuncInner implements Hidden {
              private int i = 11;
              public int value () { return i;}
        }
        return new FuncInner();
   }
}
public class Test {
   public static void main(String[] args) {
        Outer out = new Outer ();
        Outer.FuncInner in2 =
              out.getInnerInstance();
        /*EROARE: FuncInner nu este vizibila */
        Hidden in3 = out.getInnerInstance();
        System.out.println(in3.value());
   }
}
```

Clase interne în metode - locale

- Singurii modificatori care pot fi aplicați acestor clase sunt abstract sau final

- nu pot folosi variabilele declarate în metoda respectivă şi nici parametrii metodei

- Pentru a le putea accesa, variabilele trebuie declarate final

Explicație:

- Variabilele şi parametrii metodelor se află pe segmentul de stivă (zonă de memorie) creat pentru metoda respectivă, ceea ce face ca ele să nu existe la fel de mult cât clasa internă.
- Dacă variabila este declarată final, atunci la runtime se va stoca o copie a acesteia ca un câmp al clasei interne, în acest mod putând fi accesată şi după execuția metodei.

Clase interne în metode - Exemplu

```
public void f() {
  final Student s = new Student();
  /* s tb declarat final ca sa poata fi
  accesat din ModStudent */

  class ModStudent {
     public void modData() {
          s.name = ... // OK
          s = new Student(); // GRESIT!
     }
  }
}
```

Clase interne în blocuri - Exemplu

```
interface Hidden {
   public int value ();
}

class Outer {
   public Hidden getInnerInstance(int i) {
       if (i == 11) {
              class BlockInner implements Hidden {
                     private int i = 11;
                     public int value() {
                            return i;
                     }
              }
              return new BlockInner();
       } // if
       return null;
   }
}
```

Clase interne în blocuri

Observații:

- Definirea clasei interne în cadrul unui bloc *if* nu înseamnă că declarația va fi luată în considerare doar la rulare, în cazul în care condiția este adevarată!
- Semnificația declarării clasei într-un bloc este legată strict de vizibilitatea acesteia – doar in blocul *if*!
- La compilare, clasa va fi creată indiferent care este valoarea de adevăr a condiției *if*!

Clase anonime

- Clasa anonimă = clasă internă locală fără nume folosită pentru instanțierea unui singur obiect.

- sunt foarte utile în crearea unor obiecte ce implementează o anumită interfață sau extind o anumită clasă abstractă sau concreta.

```
metoda( new Interfata sau Clasa( ) {
   // Implementarea metodelor interfetei
   }
);
```

Clase anonime

- **Exemplu:**

```
if (args.length > 0) {
  final String extensie = args[0];
  lista = director.list(new FilenameFilter(){
          /*Clasă internă anonimă, creează un
          obiect al unei clase anonime ce
          implementeaza FilenameFilter! */
      public boolean accept (File dir, String
                                        nume){
          return (nume.endsWith("." +
                                   extensie) );
      } // metoda accept
  } //clasa anonima
 ); // apel list
} //if
```

Clase anonime

- Pot extinde o clasă sau să implementeze o singură interfață
- nu pot face ambele ca la clasele ne-anonime (interne sau nu)
- nici nu pot să implementeze mai multe interfețe.
- Nu pot avea constructori!
- Clasa este creată cu constructorul implicit!
- Dacă dorim să invocăm un alt constructor al clasei de bază -> transmiterea parametrilor către constructorul clasei de bază direct la crearea obiectului de tip clasă anonimă:

  new Persoana(“Mihai") { ... }

- am instanțiat o clasa anonimă, ce extinde clasa **Persoana**, apelând constructorul clasei de bază cu parametrul “*Mihai*”.

Exemplu utilizare clase interne

- Listarea fişierelor din directorul curent care au anumită extensie primită ca argument. Dacă nu se primeşte nici un argument, vor fi listate toate.

- **Varianta 1: doua clase diferite – trebuie sa transmita datele de la una la alta:**

```
import java .io .*;

class Listare {
  public static void main(String [] args){
      try {
            File director = new File (".");
            String[] list ;
```

Interfaţa FilenameFilter: exemplu

```
        if ( args.length > 0)
          list = director.list(new Filtru(args[0]));
        else
          list = director.list ();
        System.out.println( list);
        } catch ( Exception e) {
              e. printStackTrace (); }
    }
}

class Filtru implements FilenameFilter {
   String extensie ;
   Filtru ( String extensie ) {
       this.extensie = extensie ;
   }
   public boolean accept(File dir, String nume){
       return ( nume.endsWith("." + extensie));
   }
}
```

Varianta cu clasa interna membru

```
class Listare {
   String extensie;
   public static void main ( String [] args ) {
        try {
             File director = new File (".");
             String [] list ;
             if ( args . length > 0){
                  extensie=args[0];
                  list = director.list(new
                                 Filtru());
             }
             else list = director.list();
             System.out.println(list);
        } catch ( Exception e) {
             e.printStackTrace (); }
   } //main
   class Filtru implements FilenameFilter {
       public boolean accept(File dir, String
   nume){
       return ( nume.endsWith("." + extensie ) );
       }
   }
}
```

Varianta cu clasa interna locala in bloc

```
class Listare {
 public static void main ( String [] args ) {
  try {
    File director = new File (".");
    String [] list ;
    if ( args.length > 0){
      final String extensie=args[0];
      class Filtru implements FilenameFilter {
        public boolean accept(File dir, String
                                              nume){
          return (nume.endsWith("."+extensie));
        }
      }
      list = director.list( new Filtru());
   }
   else
      list = director.list();
   System.out.println(list);
  } catch ( Exception e) {
    e. printStackTrace (); }
 }
}
```

Varianta cu clasa anonima

```
class Listare {
 public static void main ( String [] args){
  try {
    File director = new File (".");
    String[] list ;
    if (args.length > 0){
      final String extensie=args[0];
      list = director.list(New FilenameFilter(){
        public boolean accept(File dir, String
                                              nume){
         return (nume.endsWith("." + extensie));
        }
      });
    }
    else
      list = director.list();
     System.out.println(list [i]);
  } catch ( Exception e){ e.printStackTrace();}
 }
}
```

Utilizarea claselor interne

- Pentru o clasă care:
  - să nu fie accesibilă din exterior sau
  - nu mai are utilitate în alte zone ale programului
- Implementăm o anumită interfață şi vrem să întoarcem o referință la acea interfață, ascunzând în acelaşi timp implementarea.
- Dorim să folosim/extindem funcționalități ale mai multor clase -> Putem defini clase interioare. Acestea pot moşteni orice clasă şi au, în plus, acces la clasa exterioară.
- Implementarea unei arhitecturi de control, marcată de nevoia de a trata evenimente într-un sistem bazat pe evenimente.
- Exemplu: Swing - GUI (graphical user interface)

Nested classes vs *inner classes*

Inner classes:

- datorită relației pe care o au cu clasa exterioară (depind de o instanță a acesteia).
- Cuprind:
  - Clasele interne membru
  - Clasele interne anonime
  - Clasele interne (locale) blocurilor şi metodelor

Nested classes :

- definirea unei clase în interiorul altei clase
- cuprinde atât inner classes cât şi clasele statice interne.
- clasele statice interne: static nested classes (nu static inner classes).

Clase incluse statice

- Clasele incluse pot avea modificatorul static (clasele exterioare nu pot fi statice!)
- Putem obține o referință către o clasă inclusa statică fără a avea nevoie de o instanță a clasei exterioare!

- Diferența clase incluse statice şi cele nestatice: clasele nestatice țin legătura cu obiectul exterior în vreme ce clasele statice nu păstrează această legătură.

Pentru clasele incluse statice:

- nu avem nevoie de un obiect al clasei externe pentru a crea un obiect al clasei incluse
- nu putem accesa câmpuri nestatice ale clasei externe din clasă inclusa (nu avem o instanță a clasei externe)

Exemplu

```
class Outer {
   public int data= 9;

   class NonStaticInner {
       private int i = 1;
       public int value() {
          return i + data;
          //Outer.this.data;
          // OK, acces membru clasă exterioară
       }
   }

    static class StaticNested {
       public int k = 99;
       public int value() {
          k += data; // EROARE, membru nestatic
          return k;
       }
   }

}
```

Exemplu

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

      Outer out = new Outer ();

      Outer.NonStaticInner nonSt = out.new
                        NonStaticInner();
      //CORECT pt o clasa nestatica

      Outer.StaticNested st2 = new
                        Outer.StaticNested();
      //CORECT pt o clasa statice

      Outer.StaticNested st = out.new
                        StaticNested();
      //INCORECTA a clasei statice
  }
}
```

Clasa File
Clasa RandomAccessFile

Programare Orientată pe Obiecte

Clasa File (1)

- Clasa *File* nu se referă doar la un fişier ci poate reprezenta fie un fişier anume, fie mulțimea fişierelor dintr-un director.
- Utilitatea clasei *File* constă în furnizarea unei modalități de a abstractiza dependențele căilor şi numelor fişierelor față de maşina gazdă
- Oferă metode pentru testarea existenței, ştergerea, redenumirea unui fişier sau director, crearea unui director, listarea fişierelor dintr-un director, etc.
- Constructorii fluxurilor pentru fişiere acceptă ca argument obiecte de tip *File*:
  - `File f = new File("fisier.txt");`
  - `FileInputStream in = new FileInputStream(f);`
  - `String[] list()`
  - `File[] listFiles()`
  - `String getAbsolutePath()`

Clasa File (2)

- Listare fişiere dintr-un director dat, cu indentare, recursiv, în subdirectoare

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

class Dir{

  public static void main (String arg[])
  throws IOException {

     String dname =".";
     if (arg.length ==1)
          dname=arg[0];
     Dir d = new Dir();
     d.dirlist(new File(dname)," ");
  }
```

Clasa File (3)

```
public void dirlist (File d, String sp)
                    throws IOException {

 String [] files=d.list();
    //lista numelor din obiectul d

 if (files == null ) return;
 String path = d.getAbsolutePath();
    //calea completa spre obiectul d

 for(int i=0;i<files.length;i++){
  File f=new File(d+"\\"+files[i]);
  System.out.println(sp+path+"\\"+
                                    files[i]);
  if (f.isDirectory())
     dirlist (f, sp+" ");
 }
}
}
```

Clasa RandomAccessFile

- permite accesul nesecvențial (direct) la conținutul unui fişier;
- este o clasă de sine stătătoare, subclasă directă a clasei Object;
- se găseşte în pachetul java.io;
- oferă metode de tipul writeX/readX– atentie la ele!;
- permite atât citirea cât şi scriere din/în fişiere cu acces direct;
- permite specificarea modului de acces al unui fişier (read-only, read-write).

```
RandomAccessFile f1 = new
  RandomAccessFile("fisier.txt", "r");
  //deschide un fisier pentru citire

RandomAccessFile f2 = new
  RandomAccessFile("fisier.txt", "rw");
  //deschide un fisier pentru scriere si
  citire
```

Clasa RandomAccessFile (2)

- Program pentru afişarea pe ecran a liniilor dintr-un fişier text, fiecare linie precedată de numărul liniei şi de un spaţiu.

```
import java.io.*;
class A{

  public static void main(String arg[])
                          throws IOException{
      RandomAccessFile raf=new
                  RandomAccessFile( arg[0],"r");
      String s;
      int i=1;
      while( (s=raf.readLine())!=null){
            System.out.println(i+" "+s);
            i++;
      }
      raf.close();
  }
}
```