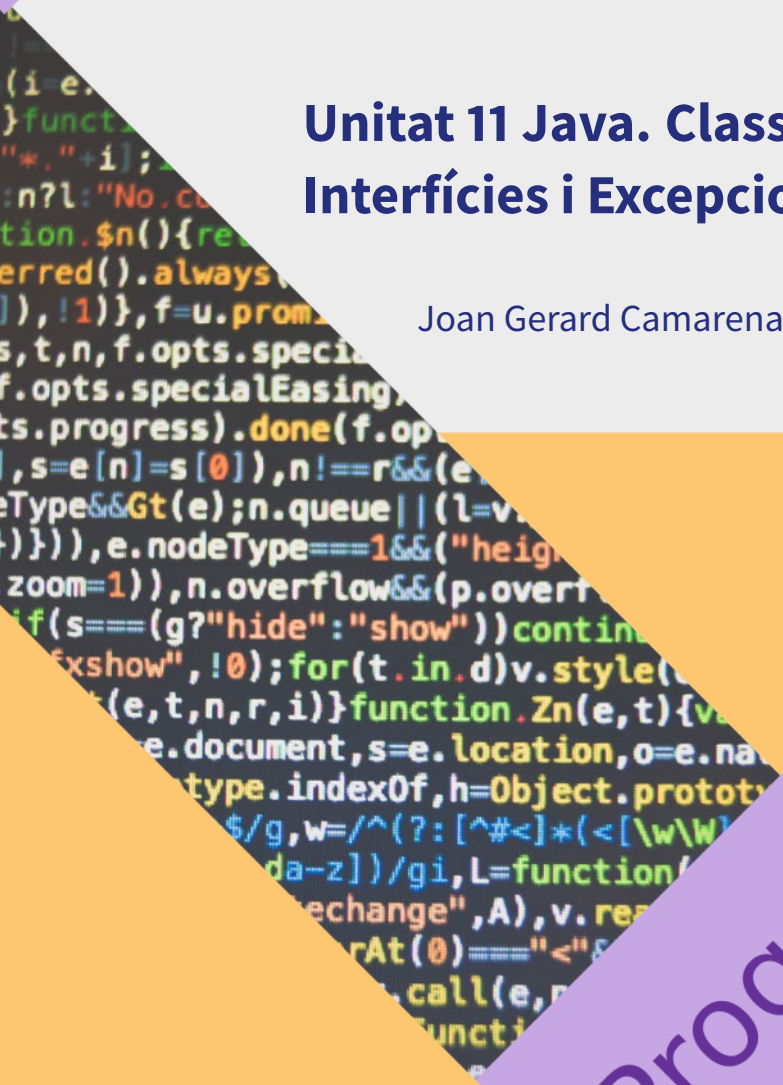
Curs 2019-2020

Unitat 11 Java. Classes Abstractes, Interfícies i Excepcions

Joan Gerard Camarena Estruch



Programació

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

1. Classes Abstractes

Suposem un esquema d'herència on tenim una classe `Professor`, de la qual hereten `Interi` i `Definitiu`, i assumim que tot professor, o és `Interi` o és `Definitiu`.

El sentit de la superclasse, és la de unificar atributs i mètodes (treure factor comú). D'altra banda, com que tot Professor, és d'una de les dos subclasses, vol dir que no crearem cap objecte de la classe Professor, sinó de les seues hereves.

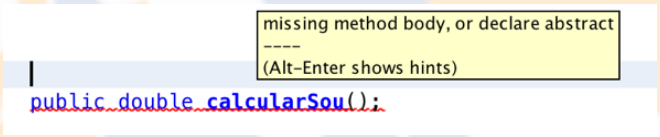
A més a més, poden fer-se coses de manera distinta en un Interí i un Definitiu, com per exemple calcular el sou. És a dir, amb un Professor no sabem com es calcula el sou, això ho sabrem si és Definitiu o Interí (però si sabrem que d'un Professor, indistintament del tipus que és, hem de calcular un sou).

Les classes abstractes solen implementar-se en les parts altes dels arbres d'herencia, indicant que són classe que **no** s'instanciaran objectes d'elles però si es derivaran.

D'altra banda, a les parts baixes dels arbres d'herència trobarem les classes **`final`**, com a indicatiu de classes de les quals ja no podem heretar.

1.1 Creació de classes abstractes

De la classe Professor no se com es calcula el sou, per tant deixem la definició sense la implementació. Els IDE es queixaran mostrant un missatge com segueix:



llavors el mètode per poder deixar-lo en blanc deu declarar-se com **`abstract`** quedant la classe com es veu:

```java
public abstract class Professor {
    protected String nom;
    protected int numero;
    protected double souBase;

    public Professor(String nom, int numero, double souBase) {
        this.nom = nom;
        this.numero = numero;
        this.souBase = souBase;
    }

    public String getNom() {
        return nom;
    }

    public int getNumero() {
        return numero;
    }

    public double getSouBase() {
        return souBase;
    }

    public abstract double calcularSou();

}
```

Això provoca que la classe queda marcada com **`abstract`**també, i per tant no podrem instanciar objectes de la mateixa (no podrem fer un **`new`** d'un `Professor`).

En el moment que creem una classe hereva de `Professor`, com que és abstracta, el IDE en ho recorda que:

- Hem d'implementar els mètodes abstractes, o
- Hem de fer la classe **`abstract`**

```java
public class Interi extends Professor{

    public Interi(String nom, int numero, double souBase) {
        super(nom, numero, souBase);
    }

    @Override
    public double calcularSou() {
        return souBase;
    }


}
```

```
public  class Definitiu extends Professor{
private int antiguetat;
private double souTrieni;

    public Definitiu(int antiguetat, double souTrieni, String nom,
            int numero, double souBase) {
        super(nom, numero, souBase);
        this.antiguetat = antiguetat;
        this.souTrieni = souTrieni;
    }

    @Override
    public double calcularSou() {
        return souBase+(antiguetat/3)*souTrieni;
    }

}
```

> Com pot observar-se, i amb tot el sentiti, cada classe implementa el mètode `calcularSou()` de manera distinta.

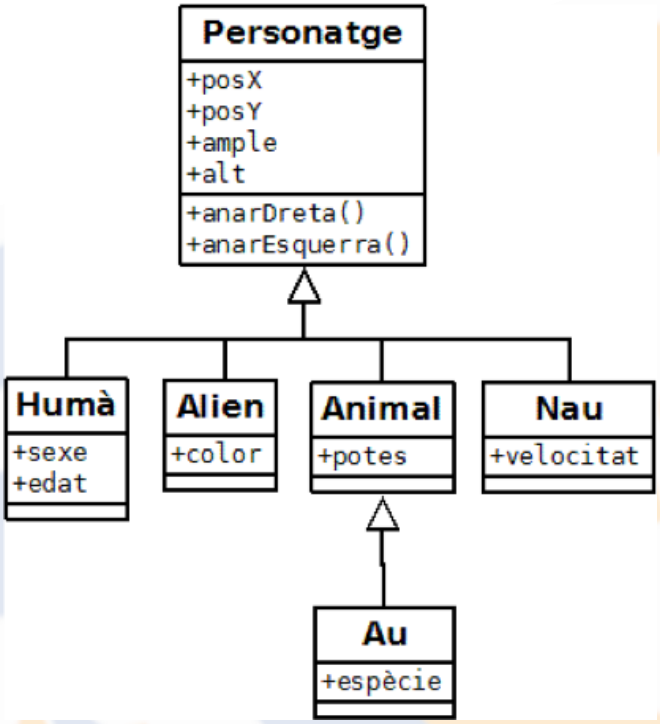
Un exemple d'ús seria com segueix. Adonar-se que no pot instanciar-se cap `Professor`, però si variables referència de `Professor` que apuntem a objectes `Interi` o a `Definitiu`.

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

            Professor p;
            p=new Interi("Pere", 123, 950);
            System.out.println(p.calcularSou());

            p= new Definitiu(6, 34.5, "Anna", 234, 950);
            System.out.println(p.calcularSou());


                        Professor is abstract; cannot be instantiated

                        The assigned value is never used
                        ----
                        (Alt-Enter shows hints)
            p=new Professor("Pere", 445, 950);

        }
}
```

```
utput - Tema26PRG (run)
run:
950.0
1019.0
```

2. Interfícies

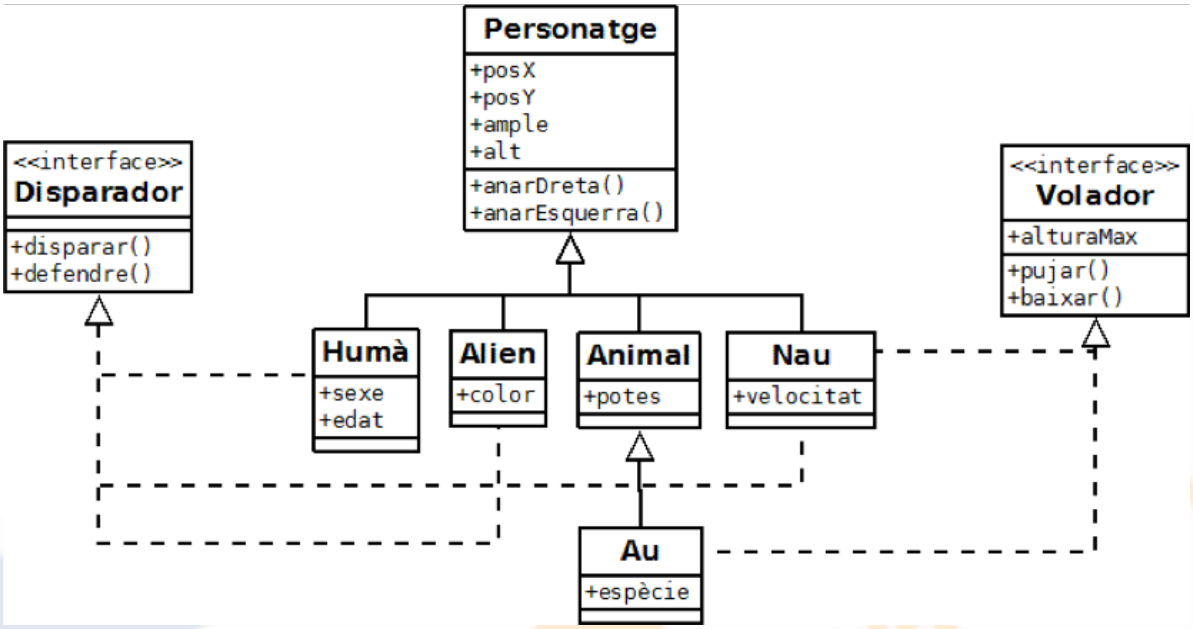
En un videojoc de marcianets hi ha molts personatges (objectes) per la pantalla. Tots seran de la classe `Personatge`, però són de tipus diferents.



2.1 Justificació i exemple

Però ara volem completar-ho fent que algunes de les classes tinguen un comportament en comú. Per exemple, si volem que la classe `Nau` i la classe `Au` puguen volar, voldrem que les dos implementen uns mètodes comuns que han de tindre un personatge capaç de volar (*pujar* i *baixar*). O bé, que les classes `Nau`, `Huma` i `Alien` tinguen el comportament en comú d'un personatge que siga disparador (*disparar* i *defendre*).

La forma ideal de fer-ho seria fer una classe `Volador` amb eixos mètodes, i altra `Disparador` amb els altres mètodes. Però en Java no podem fer que una classe siga filla de 2 classes, i aquestes classes ja hereten de `Personatge`. Per a això estan les interfícies.



Per a fer això, ens definirem una interfície Volador (i altra Disparador), on posarem els mètodes que hauran d'implementar les classes *voladores* (i *disparadores*). Els mètodes de la interfície no estaran implementats, sols la capçalera a l'igual que les classes abstractes

```
public interface Volador {
    int alturaMax = 100;
    void pujar();
    void baixar();
}
```

```
public interface Disparador {

    void disparar();
    void defendre();
}
```

Notar que:

- Els mètodes no s'implementen
- Els atributs son **`static`** i **`final`**, encara que no s'indique.

Implementcaió de la classe `Nau`:

```
1  public class Nau extends Personatge implements Volador, Disparador{
2      int velocitat;
3
4      // IMPLEMENTACIÓ DELS MÉTODES DE LA INTERFÍCIE Volador:
5      @Override
6      public void pujar() {
7          this.posY += 3;
8          if (this.posY > Volador.alturaMax) this.posY = Volador.
               alturaMax;
9          this.velocitat++;
10     }
11
```

```
12      @Override
13      public void baixar() {
14          this.posY -= 3;
15          if (this.posY < 0) this.posY = 0;
16          this.velocitat--;
17          if (this.velocitat < 0) this.velocitat = 0;
18      }
19
20      // IMPLEMENTACIÓ DELS MÉTODES DE LA INTERFÍCIE Disparador:
21      @Override
22      public void disparar() {
23          System.out.println("Pinyou, pinyou");
24      }
25
26      @Override
27      public void defendre() {
28          System.out.println("Augh!");
29      }
30  }
31
32  public class main {
33    public static void main(String[] args){
34       Nau n1 = new Nau();
35       Volador v1 = new Nau();     // puc crear referencies Voladors,
36       Volador v2 = new Au(); // però no objectes
37       ArrayList <Volador> llistaVoladors = new ArrayList<>();
38       llistaVoladors.add(n1);
39       llistaVoladors.add(v2);
40      ...
41  }
```

2.2 Creació d'interfícies

Recordem que podem fer classes amb:

- *Mètode abstracte*: mètode on no consta la implementació. I per tant:
- *Classe abstracta*: classe que té algun mètode abstracte. No es pot instanciar.

Les classes filles d'una classe abstracta estan obligades a implementar eixos mètodes abstractes, o bé tornar a declarar-los com a abstractes (i, per tant, eixa altra classe també serà abstracta).

Vist això, una interfície és com una **classe abstracta pura** (no té implementat cap mètode) amb l'avantatge que una classe pot implementar (*ser filla de*) moltes interfícies (però sols pot extendre d'una classe).

> Una interfície també pot extendre d'una altra interfície.

Interfície conjunt de mètodes sense implementar que hauran d'implementar aquelles classes que vullguen comportar-se així. També pot incloure constants.

És per això, que en la literatura de programació es diu que implementar una interfície és comportar-se d'una determinada manera. Per això si una classe implementa diverses interfícies estem davant d'objectes que poden tenir diversos comportaments.

Utilitats:

- Simular herència múltiple (ja que una classe només pot tindre una superclasse però pot implementar moltes interfícies).
- Obligar a que certes classes utilitzen els mateixos mètodes (noms i paràmetres) sense estar obligades a tindre una relació d'herència.
- Sabent que una classe implementa una determinada interfície, podrem usar els seus mètodes perquè ja sabrem què fan (ens dóna igual com estiguen implementats).
- Definir un conjunt de constants

3 Interficies vs Classes Abstractes

	Classes	Interfícies
Quantitat de pares d'una classe	1	n
Poden instanciar-se?	Si	No

	Classes Abstractes	Interfícies
Poden implementar-se mètodes	Si	No
Poden definir-se atributs	Si	Constants (`final`)

Per concretar:

1. Les interfícies són regles (ja que s'imposa el fer una implementació per a elles i que no es pot evitar) i funciona com un contracte entre els diferents equips de desenvolupament de programari.
2. Les interfícies donen la idea del que cal fer, però no com es va a fer. La implementació depèn completament de desenvolupador, seguint les regles donades (mitjançant la signatura del contracte).
3. Les classes abstractes poden contenir declaracions abstractes, implementacions concretes, o ambdues coses.
4. Les declaracions Abstract són les regles que s'han de seguir i les implementacions concretes són com directrius (podeu fer-les seguir o canviar-les).

4. Interfícies `Comparable` i `Comparator`

4.1 Justificació

Per a ordenar un vector o un ArrayList d'enters, podem fer servir mecanismes ja implementats i estudiats de les llibreries de classes de Java:

```
1 int [] edats = {4,7,3,6,9};
2 Arrays.sort(edats); // Cal importar java.util.Arrays
3 System.out.println(Arrays.toString(edats));
4
5 ArrayList <Integer> edats2 = new ArrayList();
6 edats2.add(4);edats2.add(7);edats2.add(3);edats2.add(6);edats2.add(9);
7 Collections.sort(edats2);   // Cal importar java.util.Collections
```

D'igual forma podríem ordenar una llista (array, ArrayList…) de String, de float, etc. Però què passa si volem ordenar una llista d'elements que no són directament ordenables (comparables), com pot ser una llista de cotxes, d'alumnes, etc? Si li aplicàrem el mètode `sort`, ens donaria error, ja que la MVJ *no sap comparar* eixos objectes.

És més si mirem quina és la sintàxi del mètode sort, ens trobem que:

> **`public static <T extends Comparable<? super T>> void sort(List<T> list)`**
> Sorts the specified list into ascending order, according to the natural ordering of its elements. **All elements in the list must implement the Comparable interface**. Furthermore, all elements in the list must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

i

> **`public static <T> void sort(List<T> list, Comparator<? super T> c)`** Sorts the specified list according to the order induced by the specified comparator. **All elements in the list must be mutually comparable using the specified comparator** (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

Per a fer que els objectes d'una classe puguen ser comparats, hem d'indicar un criteri de comparació. És a dir, cal definir quan un objecte de la classe que volem és menor que un altre, quan és major i quan és igual. Si volem establir un únic criteri d'ordenació, usarem la interfície `Comparable` però si volem establir diferents criteris d'ordenació usarem la interfície `Comparator`.

4.2 `Comparable`

L'han d'implementar les classes que vullguen establir un criteri de comparació dels seus objectes (i només un). L'únic mètode que conté, i per tant cal implementar és `int compareTo(Object obj)`

Este mètode retorna un número negatiu, un zero o un número positiu, depenent de si `this` és menor, igual o major a `obj`. Ens servirà per a comparar dos objectes pel criteri que volem. Observar que `obj` és genèric, i per tant caldrà fer un càsting en determinades ocasions.

Exemple: suposem que volem comparar (o ordenar) alumnes. Si volem que l'ordre natural dels alumnes és pel seu *codi*, farem:

```
1  public class Alumne implements Comparable{
2      int codi;
3      String nom;
4      int edat;
5      String curs;
6
7
8  // Així aconseguim que retorne
9  // un número negatiu si this és menor que obj;
10 // un número positiu si this és major que obj;
11 // o un zero si són iguals.
12     @Override
13     public int compareTo(Object obj) {
14         return this.codi - ((Alumne)obj).codi;
15     }
16 }
```

Si volem estalviar-se el càsting, hauriem d'indicar-ho d'altra manera:

```
1  public class Alumne implements Comparable <Alumne> {
2      int codi;
3      String nom;
4      int edat;
5      String curs;
6
7      @Override
8      public int compareTo(Alumne alu) {
9          return this.codi - alu.codi;
10     }
11 }
```

> Com llegir-ho: sd - Primer cas, l'alumne és comparable - Segon cas, l'alumne és comparable, però sols amb alumnes

Ara al nostre codi ja podem fer coses com:

```
    if ( alume1.compareTo(alumne2) < 0 ) { ... }

ArrayList <Alumne> llistaAlumnes = new ArrayList();
llistaAlumnes.add( new Alumne(...) );
llistaAlumnes.add( new Alumne(...) );
          ...
Collections.sort(llistaAlumnes);
System.out.println(llistaAlumnes);
```

El que acabem de veure és un altre motiu de l'ús d'interfícies: mitjançant la implementació d'interfícies tots els programadors fan servir el mateix nom de mètode i estructura formal per comparar objectes (o clonar, o altres operacions).

Imagina't que estàs treballant en un equip de programadors i has d'utilitzar una classe que ha codificat un altre programador. Si vols comparar dos objectes d'eixa classe, només veient que implementa la interfície Comparable, ja saps quins mètodes pots usar sense saber com està implementat.

Això facilita el desenvolupament de programes i ajuda a comprendre'ls, sobretot quan intervenen centenars de classes diferents.

4.3 Comparator

Amb la interfície `Comparable` podíem comparar (ordenar) alumnes per un criteri establert: el codi de l'alumne. Però i si decidim ordenar-los pel nom, o pel curs, etc?

Per a fer que els objectes d'una classe puguen ser comparats per diversos criteris, per cada criteri caldrà crear un classe especial que implemente una interfície anomenada `Comparator`, on definirem el mètode `compare` (no `compareTo`), al qual se li passen com a paràmetre els dos objectes a comparar i retornarà un valor negatiu, zero o positiu, semblant al mètode `compareTo`.

Exemple: Ja hem fet que l'alumne siga comparable (pel codii). Ara anem a crear **comparadors** de *nom* i d'*edat*

```
import java.util.Comparator;
class ComparadorAlumneNom implements Comparator<Alumne> {
    @Override
    public int compare(Alumne p1, Alumne p2){
        return p1.getNom().compareTo(p2.getNom());
   }
}
class ComparadorAlumneCurs implements Comparator<Alumne>{
    @Override
    public int compare(Alumne p1, Alumne p2){
        return p1.getCurs() - p2.getCurs();
    }
}
```

Fixar-se que:

- Hem creat dos classes, que son comparadores d'Alumnes,
- Cada classe sols implementa el mètode `compare`
- El mètode retorna un enter, a l'igual que `compareTo`

Ara, per a fer-ho servir, hem d'utilitzar el segon mètode `sort`, que es passa una llista i un comparador:

```
1  if ((new ComparadorAlumneNom()).compare(a1, a2) < 0) { ... }
2
3  // o bé pel criteri del curs:
4  if ((new ComparadorAlumneCurs()).compare(a1, a2) < 0) { ... }
5
6  /* O bé, ordenar una llista d'alumnes (array, ArrayList...) amb el sort
       , com abans, però passant-li també l'objecte que té el criteri de
       comparació: */
7
8  Collections.sort(llistaAlumnes, new ComparadorAlumneNom() );
9
10 // o bé pel criteri del curs:
11 Collections.sort(llistaAlumnes, new ComparadorAlumneCurs() );
```

Llavors estam davant d'una potent ferramenta, ja que amb implementacions mínimes podem fer servir totes les llibreries d'ordenació que ens dona Java.

5. Interfície `Iterable` i `Iterator`

5.1 Justificació i ús

Fins ara quan hem recorregut llistes hem fet ús dels bucles **`for`** o `foreach`. Ocòrre que de vegades ens proporcionaran una llista que sabem que conté una col·lecció d'elements, sense saber com estan guardats (i més encara sense accedir a ells, donat que probablement seran privats). En aquests casos es fa que aquestes classes siguin *iterables*, és a dir, que d'alguna manera oferisquen un mecanisme de recorregut dels seus elements. Aquest mecanisme és un *iterador*. Un `Iterator` és un objecte que disposa de:

- **`public boolean`** `hasNext()` → retorna si hi ha un altre element o no a recòrrer.
- **`public`** `E next()` → retorna el següent element (*E* : classe que vullgam).
- **`public void`** `remove()` → elimina l'últim element retornat.

```
Iterator<String> it = llista.iterator();
while (it.hasNext()) {
    String nom= it.next();
    if (nom.equals("Pep"))
        it.remove();
}
```

> Nota: el remove no podem aplicar-lo en un `foreach`

Amb les interfícies `Iterator` i `Iterable` podrem recórrer una col·lecció. Utilitats d'estes interfícies:

- Recórrer una col·lecció mentre esborrem alguns dels seus elements.
- Recórrer una col·lecció sense saber com està implementada.
- Recórrer diferents tipus de col·leccions de la mateixa forma
- Recórrer una mateixa col·lecció per diferents recorreguts.

Exemple amb la classe ArrayList, que implementa la interfície:

```
ArrayList <Persona> llistaPersones = new ArrayList();
Iterator <Persona> it = llistaPersones.iterator();
...
Persona p;  // Objecte temporal
while (it.hasNext()){
      p = it.next();
      if (p.getEdat() < 18) {
        System.out.println("És menor. L'esborrem");
        it.remove();
}
```

5.2 Implementació de `Iterator`

Anem a fer una classe `Departament`, que conté una col·lecció de `Empleat`. La flexibilitat és que, un cop implementada, podrem recòrrer el departament sense saber que hi ha dins ni com estan guardats els Empleats:

Classe Empleat:

```java
/**
 * Classe que conté els elements de la llista. Res d'especial
 * @author joange
 */
public class Empleat {
    private String nom;
    private String carrec;
    public Empleat(String nom, String carrec) {
        this.nom = nom;
        this.carrec = carrec;
    }

    public String getNom() {
        return nom;
    }

    public String getCarrec() {
        return carrec;
    }

    @Override
    public String toString() {
        return "Emp{" + "nom=" + nom + ", càrrec=" + carrec + '}';
    }
}
```

Classe Departament:

```java
/**
 * Classe que conté la llista
 * @author joange
 */
public class Departament {

    private String nom;
    private Empleat[] llistaEmpleats = new Empleat[100];
    private int qEmpl = 0;

    Departament(String nom) {
        this.nom = nom;
    }

```

```
15      public void add(String nomEmpleat, String carrec) {
16          llistaEmpleats[qEmpl++] = new Empleat(nomEmpleat, carrec);
17      }
18
19
20      /**
21      * Classe interna. Es defineix dins de Departament per accedir
22      * directament als membres del Departament.
23      */
24      protected class IteradorDEmpleats implements Iterator<Empleat> {
25
26          private int posicio = 0;
27
28          @Override
29          public boolean hasNext() {
30              return posicio < qEmpl;
31          }
32
33          @Override
34          public Empleat next() {
35              return llistaEmpleats[posicio++];
36          }
37
38          @Override
39          /**
40           * Eliminarem el anterior a la posició actual. Que és el que
                 hem recuperat amb next
41           */
42          public void remove() {
43              int eliminar = posicio - 1;
44              if (eliminar < 0) {
45                  return;
46              }
47              if (eliminar < qEmpl - 1) {
48  System.arraycopy(llistaEmpleats, eliminar + 1, llistaEmpleats,
        eliminar, qEmpl - 1);
49              }
50              qEmpl--;
51          }
52      }
53
54      /**
55      * @return Iterador per a recorrer els empleats del departament.
56      * Retorna un Iterador de la clase interna
57      */
58      public Iterator<Empleat> iterador() {
59          return new IteradorDEmpleats();
60      }
61
62      @Override
63      public String toString() {
```

```
64          String res = "";
65          for (int i = 0; i < qEmpl; i++) {
66              res += llistaEmpleats[i] + "\n";
67          }
68          res += "\b";
69          return res;
70      }
71  }
```

Finalment, el codi del programa principal on fem servir les classes Empleat i Departament amb el seu Iterador. Per exemple, esborrarem de la llista els empleats que tenen de càrrec *no res*.

```
1  public static void main(String arg[]) {
2      Departament dep = new Departament("Informàtica");
3      Iterator<Empleat> it;
4      Empleat empl;
5      dep.add("Marc", "no res");
6      dep.add("Pep", "programador");
7      dep.add("Alfred", "no res");
8      dep.add("Maria", "analista");
9
10     it = dep.iterador();
11     while (it.hasNext()) {
12         empl = it.next();
13         if (empl.getCarrec().equals("no res"))
14         {
15             it.remove();
16         }
17     }
18     System.out.println("Empleats del departament:\n" + dep.toString());
19 }
```

5.3 Implementació de `Iterable`

Si definim una classe amb una llista que volem que siga utilitzada amb un iterador, és convenient indicar que la classe és *iterable*. És a dir: que es pot recórrer amb un iterador que ja té definit. Això s'aconsegueix fent que la nostra classe implemente la interfície `Iterable`, que només té el mètode `iterator()`.

En l'exemple que hem vist abans, només caldria indicar que la classe Departament implementa la interfície `Iterable` i substituir el mètode `iterador()` per `iterator()`:

```
1  public class Departament implements Iterable {
2      private String nom;
3      private Empleat[] llistaEmpleats = new Empleat[100];
4
5      @Override
```

```
6       public Iterator<Empleat> iterator() {
7           return new IteradorDEmpleats();
8       }
9
10      protected class IteradorDEmpleats implements Iterator<Empleat> {
11          @Override
12          public boolean hasNext() { ... }
13
14          @Override
15          public Empleat next() { ... }
16
17          @Override
18          public void remove() { ... }
19      }
20  }
```

Resumint: per a implementar la interfície `Iterable` hem de sobreescriure el mètode `iterator()`, i per a això hem de poder tornar un objecte `Iterator`, la qual cosa aconseguim creant una classe interna que implementa la interfície `Iterator`.

> Nota: si volem usar el bucle for-each en una classe haurà d'implementar `Iterable`.

6. Interfície `Cloneable`

6.1 Justificació

Recordem que per a copiar un objecte a un altre no podem fer servir l'operador d'assignació (=) ja que tindríem només un objecte però amb dos referències a ell. El que vegèrem de fer és un mètode per a copiar atribut a atribut.

La classe `Object` ja té eixe mètode, anomenat `clone()`, que retorna un `Object` idèntic. La forma d'usar el clone seria així:

```
1 Cotxe c1 = new Cotxe("Seat", 10);
2 Cotxe c2 = c1.clone();
```

Però per a poder invocar eixe mètode estem obligats a implementar-lo en la nostra classe, ja que en la classe `Object`està definit com a **`protected`**.

6.2 Implementació de `clone()`

Per a implementar el `clone()` en la nostra classe `Cotxe` ho podem fer de dos formes distintes: invocant al `clone()` de la classe `Object` o sense invocar-lo:

1. La còpia la fa la pròpia classe (no invoca a **`super`**`.clone()`):

- Hem de reservar memòria per al nou objecte
- Hem de copiar atribut a atribut al nou objecte

2. La còpia la fa el pare de la classe, `Object` (invoca a **`super`**`.clone()`):

- No cal reservar mèmòria
- No hem de copiar atribut a atribut
- Cridarem a **`super`**`.clone()` (dins d'un *try-catch*)
- La classe ha de tindre el **`implements`** `Cloneable`.

Exemple de 1 La còpia la fa la propia classe :

```
1 public class Cotxe {
2     String matr;
3     int anys;
4 
5     public Cotxe(String matr, int anys){
6       this.matr = matr;
7       this.anys = anys;
8     }
```

```
9
10      @Override
11      public Cotxe clone(){
12          Cotxe clon = new Cotxe();
13          clon.matr = this.matr;
14          clon.anys = this.anys;
15          return clon;
16      }
17
18      @Override
19      public Cotxe clone(){
20              return new Cotxe(this.matr, this.anys);
21      }
22  }
```

> Els mètodes `clone()`, un dels dos, el que més ens agrade.

Exemple de 2: La còpia la fa `Object`:

```
1  public class Persona implements Cloneable{
2      String nom;
3      int edat;
4      Cotxe cotxe;
5
6      public Persona(String nom, int edat, Cotxe cotxe) {
7          this.nom = nom;
8          this.edat = edat;
9          this.cotxe = cotxe;
10     }
11
12     @Override
13     public Persona clone(){
14         Persona clon=null;
15         try {
16           // Fem el clon amb Object, i llavors el convertim a Persona
17           clon = (Persona) super.clone();
18           // Com Cotxe és un objecte l'he de clonar ad-hoc
19           clon.cotxe = this.cotxe.clone();
20           } catch (CloneNotSupportedException ex) {
21             System.out.println("No es pot duplicar");
22         }
23         return clon;
24     }
25 }
```

Eixa interfície (`Cloneable`) és especial perquè no té cap mètode. Per tant, per a què serveix? Com clonar un objecte pot ser *perillós*, serveix per a dir-li al `clone()` de la classe `Object` (quan és invocat per la nostra classe) que estem d'acord que faça una còpia camp per camp. Si el `clone()` d'`Object` comprova que la nostra classe no implementa `Cloneable`, botarà l'excepció `CloneNotSupportedException`

.

7. Excepcions

7.1 Error vs Excepció

Un error dins d'un programa és una errada que obliga a aturar-lo de manera inesperada. Quan això passa diem que el programa s'ha penjat o bloquejat. Als llenguatges de programació moderns això afortunadament ha canviat, i l'error s'ha redefinit per una *excepció*.

Un excepció la podem definir com una situació durant l'execució d'un programa que deté el fluxe d'execució del mateix. En eixe moment eixa excepció deu ser tractada (per intentar solucionar la situació que l'ha provocada):

- L'excepció es controla → el programa pot seguir treballant amb normalitat
- L'excepció no es controla → el programa o funció on s'ha donat l'excepció la propaga a qui ha invocat el programa (SO) i aquest *pare* (qui ha invocat el mètode) és qui te que controlar la excepció (passar-se la creïlla calenta).

7.2 Captura i maneig d'Excepcions

La captura d'excepcions està format per diversos blocs de codi:

- `try{}`

És el bloc on están les instruccions que poden produir una excepció. Dins d'aquest bloc poden donsar-se diverses excepcions, però sols posarem un `try`

- `catch{}`

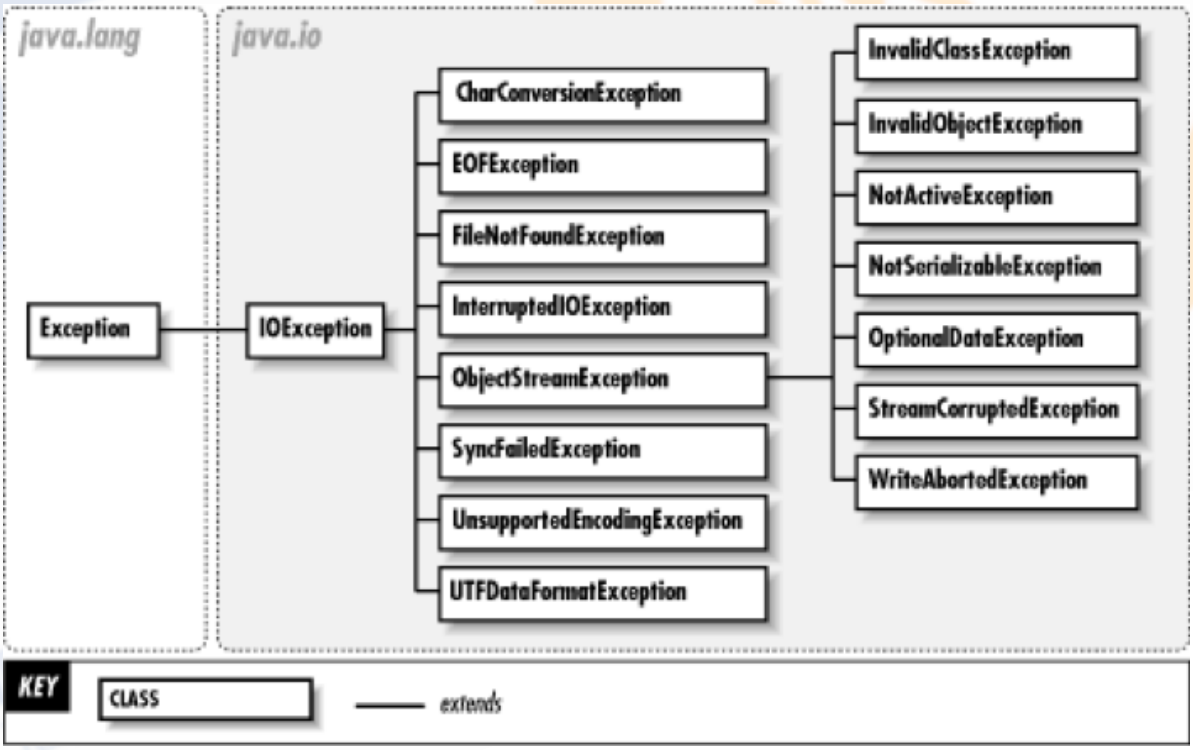
És un conjunt de blocs associat a un bloc `try{}` anterior. En ells hem de codificar que passa quan ocòrre una excepció.

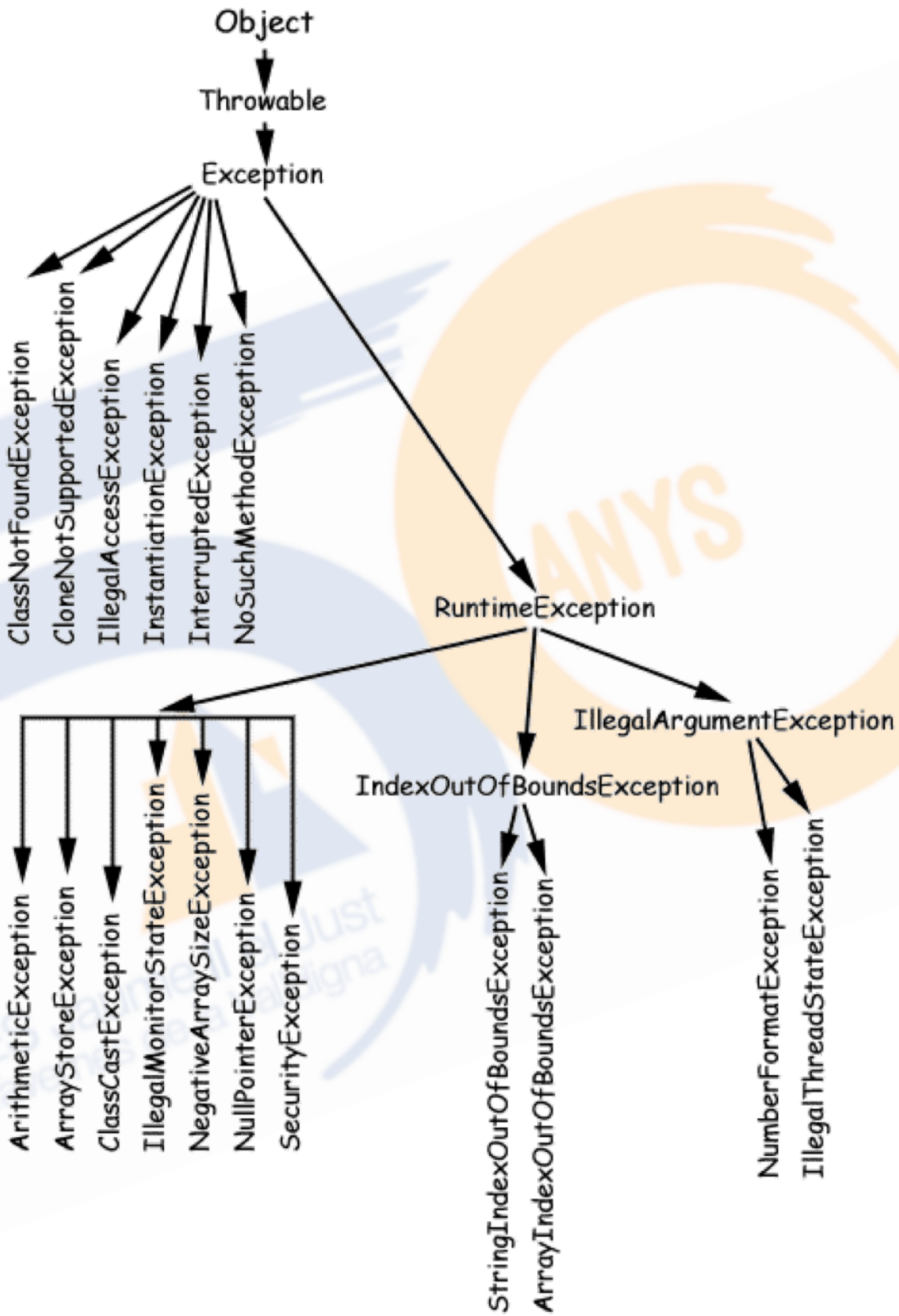
- `finally{}`

És un bloc de control. S'executa sempre, aparega o no l'excepció, i cas d'aparèixer després d'haver-la tractada. Té utilitat si volem que s'execute algun codi quan ha botat una excepció que no tenim capturada.

```java
try{
    //bloc de codi que pot donar una excepció (una o moltes)
}
// ara tants bloc catch com possibles excepcions hi han al try
catch( Exception1 ex){
    // que fem quan bota la Exception1
}
catch( Exception2 ex){
    // que fem quan bota la Exception1
}
// bloc que volem que s'execute tant si hi han com si no hi han
   excepcions
finally{
  // codi final
}
```

Quant a la captura d'excepcions, ha de fer-se de la més concreta a la més genèrica dins del bloc catch. Considerem la jerarquia d'excepcions associades a l'entrada Eixida de Java:




Object
Throwable
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalMonitorStateException
NegativeArraySizeException
NullPointerException
SecurityException
IndexOutOfBoundsException
StringIndexOutOfBoundsException
ArrayIndexOutOfBoundsException
IllegalArgumentException
NumberFormatException
IllegalThreadStateException

7.3 Creació de noves Excepcions

7.3.1 Creació d'excepcions

La creació d'excepcions és relativament senzill en Java. Hem de:

1. Definir una nova classe que herete de la classe `Exception`.
2. Crear un constructor amb un `String` com a argument.
3. Dins del constructor cridar al constructor de **`super()`**, passant-li el `String` rebut.

```
1 public class novaExcepcio extends Exception {
2     public novaExcepcio(String msg) {
3         super(msg);
4     }
5 }
```

Amb això ja tenim la nostra excepció a punt per a fer-se servir.

7.3.2 Llançament d'excepcions

Un cop creada la excepció, l'hem de fer servir o llançar-la. Al nostre codi hem de detectar quan està produint-se la situació anòmala i llançarla:

```
1 public void algunMetode() throw excepcioNova{
2     // diverses coses
3
4     if (situacioAnomala)
5       throw new excepcioNova("Descripcio del Error");
6
7     // altres coses
8 }
```

Notes:

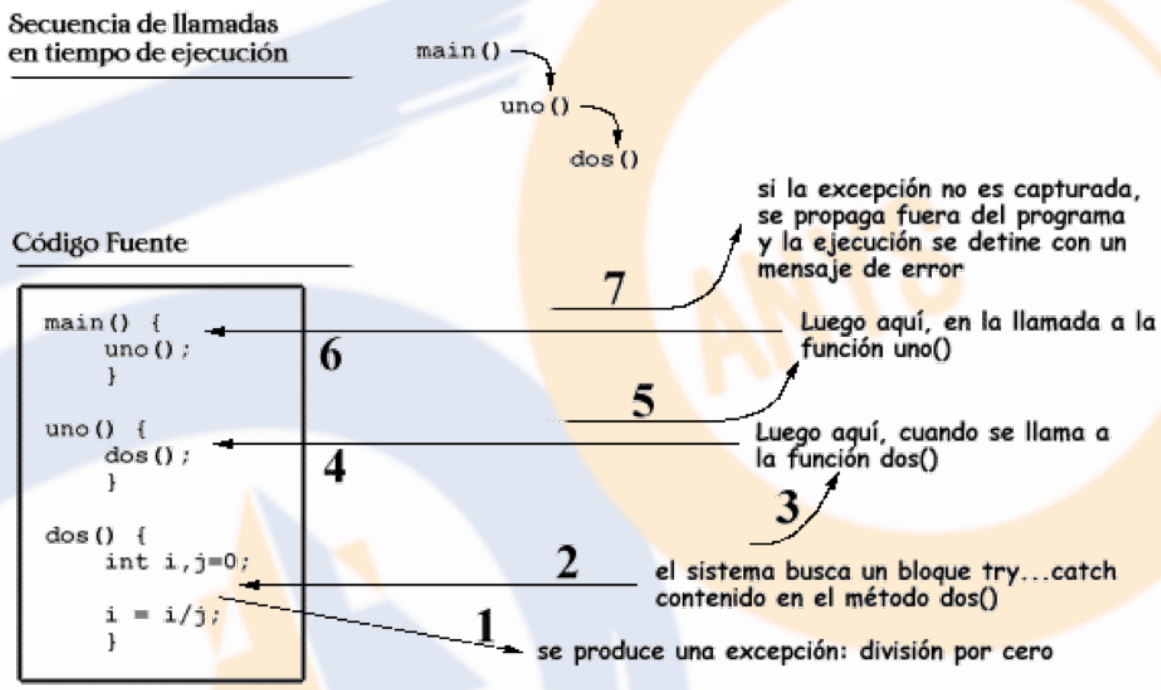
- El mètode que pot llançar la excepció deu de deixar-la eixir (**`throw`**)
- El llançament de les excepcions sempre estaran dins de sentències condicionals
- La descripció (missatge) deu de ser breu i clarificador.

Si intentem cridar a aquest mètode sense més, Netbeans donarà error, dient-nos que l'excepció no està tractada (unreported). Això és perquè el mètode la pot llançar però des d'on cridem al mètode no estem tractant la excepció.

7.4 Propagant excepcions

Si estem executant un codi que pot donar lloc a una excepció, el que hem de fer és tractar-la, tal i com hem vist abans amb un bloc **`try-catch`**.

De vegades voldrem que un mètode no tracte la excepció i que la deixe eixir al mètodes que l'han invocat o finalment fins i tot al `main()` o al Sistema Operatiu. És tan senzill com afegir a la capçalera del mètode la clàusula **`throws`** seguit de les excepcions que volem que el mètode deixe eixir. `retorn nom( parametres )throws e1,e2,e3 { }`



> Nota: Controlar que un programa que no fa el que l'usuari final vol (no funciona be), no es considera que és erroni sinó una mala implementació o disseny. Jo puc pintar una paret fantàstica de verd, però si m'havien dit que la pintara roja… És a dir, la meua execució de la paret pintada de verd és impol·luta, però no és el que estava pensat de primer hora

8. Enumeracions

Els tipus enumerats són unes classes destinades a guardar constants, que per tant són immutables. Solen definir-se en majúscules. Per exemple:

```
enum Nivell{
    MOLT_BAIX,
    BAIX,
    MITJA,
    ALT,
    MOLT_ALT
}
```

Després per a fer-les servir, haurem de fer servir la notació del punt, com si de valors es tractara.

```
Nivell n=Nivell.BAIX;

switch(nivell){
  case MOLT_BAIX:
    ...
  case BAIX:

  ...
}
```