# Material de suport Lliurament 1

## 1. Implementació de taules i tuples

Per a definir els equivalents a les Taules de tuples que heu implementat vosaltres mateixos, utilitzarem la classe de Java ArrayList.

Aquesta classe conté tots els mètodes necessaris per tal d'inserir, eliminar i accedir als elements en una taula.

**Exemple de creació d'una taula 1 (Taula homogènia):**

Creació d'una taula per a objectes de la classe Bicicleta.

```
ArrayList<Bicicleta> tauBicicletes = new ArrayList<Bicicleta>();
```

Creació d'una taula per a objectes de la classe Bicicleta reservant un espai inicial (no és un límit de capacitat).

```
ArrayList<Bicicleta> tauBicicletes = new ArrayList<Bicicleta>(mida);
```

**Exemple de creació d'una taula 2 (Taula heterogènia):**

Creació d'una taula que pot contenir objectes de qualsevol classe.

```
ArrayList  tauObjectes = new ArrayList ();
```

Creació d'una taula per a objectes de qualsevol classe reservant un espai inicial (no és un límit de capacitat).

```
ArrayList  tauObjectes = new ArrayList (numElements);
```

**Exemple d'utilització d'una taula:**

Afegir un nou element a la taula.

```
Bicicleta b=new Bicicleta();
b.demanarDades();
tauBicicletes.add(b);
```

Obtenir el nombre d'elements de la taula:

```
tauBicicletes.size();
```

Accedir a un element de la taula:

- En taules homogènies es pot accedir directament

```
Bicicleta b=tauBicicletes.get(posicioBicicleta);
```

- En taules heterogènies cal fer un cast

```
Bicicleta b=(Bicicleta) tauObjectes.get(posicioBicicleta);
```

Eliminar un element de la taula. En eliminar un element, els elements que hi havia sota d'aquest a la taula es mouen per omplir el forat deixat.

```
tauBicicletes.remove(posicioBicicleta);
```

Eliminar tots els elements de la taula.

```
tauBicicletes.clear();
```

**Exemple d'accés seqüencial als elements de la taula:**

Tot i que es podria accedir directament amb una estructura del tipus "for", en Java existeixen uns elements anomenats iteradors, que fan més eficient aquesta tasca. A continuació es mostra com iterar per la taula anterior.

Obtenir un iterador per a una taula

```
Iterator itrBicicletes = tauBicicletes.iterator();
```

Per iterar a través dels elements, s'utilitzen els mètodes "hasNext( )" i "next( )" de l'iterador. El primer ens informa de si hi ha més elements a la seqüència, i el segon ens retorna l'element actual de la seqüència i es mou al següent. Mireu l'exemple següent:

```
While(itrBicicletes.hasNext()) {
        Bicicleta b=itrBicicletes.next();
        System.out.println(b);
}
```

# 2. Utilització de Streams per fitxers

Un tipus molt utilitzat en l'entrada/sortida de Java són els Streams. Un Stream és un flux de dades. Segons si volem llegir informació d'un Stream, o afegir informació a un Stream, parlarem d'Stream d'entrada (InputStream) o Stream de sortida (OutputStream).

Un exemple d'InputStream és el System.in, el qual representa l'entrada de dades per teclat. L'exemple d'OutputStream és el System.out, el qual permet escriure informació que serà visualitzada en el monitor. En el cas dels fitxers, són elements que tant poden actuar d'entrada de dades com de sortida. A continuació es mostra com obtenir els Streams corresponents per a un fitxer.

**Exemple de creació d'Streams associats a fitxers:**

Els Streams que actuen sobre fitxers son els FileInputStream i els FileOutputStream, per entrada i sortida respectivament. Quan tractem amb fitxers, el Java en dona una classe File, que representa una ruta a un fitxer o directori del disc. Aquesta classe només representa el camí cap a un fitxer, no el fitxer en si. Fixeu-vos en l'exemple:

```
// En Windows
File fitxer=new File("C:\\Reproductor\\BibliotecaMusical.txt");
// En Linux
File fitxer=new File("~/Reproductor/BibliotecaMusical.txt ");

// Obrir un Stream d'entrada des del fitxer
FileInputStream fin=new FileInputStream(fitxer);

// Obrir un Stream de sortida cap al fitxer
FileOutputStream fout= new FileOutputStream(fitxer);
```

Sempre que s'utilitzin fitxers, és molt important tancar el fitxer quan s'ha acabat d'utilitzar. En aquest cas:

```
// Tancar l'Stream d'entrada des del fitxer
fin.close();

// Tancar l'Stream de sortida cap al fitxer
fout.close();
```

**Exemple d'utilització d'Streams:**

Els Streams que actuen sobre fitxers es poden utilitzar de la mateixa manera que utilitzàveu el System.in (amb l'Scanner) i el System.out (mètodes print).

```
// Llegim de fitxer i mostrem a pantalla
Scanner sc=new Scanner(fin);
String linia=sc.nextLine();
System.out.println("hem llegit " + linia);

// Llegim de teclat i guardem a fitxer
Scanner sc2=new Scanner(System.in);
System.out.println("Escriu una frase");
String frase=sc2.nextLine();
fout.println(frase);
```

Per saber si queda informació en l'Stream, podeu utilitzar el mètode "hasNext". Per exemple, llegir totes les línies d'un fitxer, es podria fer amb:

```
while (fin.hasNext()) {
      String linia=sc.nextLine()
      System.out.println("hem llegit " + linia);
}
```

**Exemple per guardar un objecte:**

En els exemples anteriors s'ha mostrat l'accés genèric a un Stream, tant per llegir com per guardar informació. Per guardar un objecte qualsevol, es podria fer un mètode que escrivís els valors guardats en els seus atributs en un fitxer en un ordre determinat, i per tant que es poguessin tornar a llegir en qualsevol moment. Aquest procés està automatitzat en Java, i es coneix amb el nom de "Serialització". Bàsicament una classe serialitzable és aquella que es pot representar unívocament com a una seqüència de Bytes, i això permet als seus objectes ser guardats en un fitxer o inclús enviats per una xarxa d'ordinadors. Per definir una classe com a Serializable cal declarar-la de la següent manera:

```
public class Bicicleta implements Serializable {
      .....
}
```

A més a més, tots els atributs de la classe han de ser serialitzables. Els tipus i classes de Java per defecte son serialitzables. Per escriure o llegir objectes directament, ens caldrà utilitzar els ObjectStream. A continuació es mostra el codi per utilitzar els FileStreams vistos anteriorment per a guardar i llegir un objecte de tipus Fitxer;

```
Bicicleta bici=new Bicicleta();

// Guardem el fitxer
ObjectOutputStream oos = new ObjectOutputStream(fout);
oos.writeObject(bici);

// Llegim el fitxer
ObjectInputStream ois = new ObjectInputStream(fin);
Bicicleta bici2= (Bicicleta)ois.readObject();
```

# 3. Tractament d'excepcions

Les excepcions són el mètode per defecte de tractament d'errors d'execució en Java. Una excepció és un objecte que conté informació sobre el tipus d'error, el lloc on s'ha produït i l'estat del programa en aquell moment. Analitzant aquesta informació, l'usuari o el desenvolupador poden intentar corregir el problema. Sempre que s'utilitza algun mètode que genera excepcions, és obligatori controlar aquestes excepcions.

Hi ha tres famílies d'excepcions:

1. **Excepcions controlades:** Casos excepcionals que un programa ben escrit pot preveure i per tant anticipar-se a l'error i recuperar-se. Per exemple, si demanem el nom d'un fitxer, podem preveure que el nom que ens entrin no sigui correcte, i per tant en comptes de finalitzar l'aplicació quan es generi una excepció al intentar llegir el fitxer, podem mostrar un missatge a l'usuari i demanar de nou el fitxer.
2. **Errors**: Condicions excepcionals externs a l'aplicació, i que no són previsibles. Per exemple en el cas anterior, podeu imaginar que el fitxer realment existeix, el podem obrir correctament, però hi ha un error en el disc i no podem llegir-lo correctament. El tractament d'aquests errors generalment és mostrar l'error i finalitzar el programa.

3. **Excepcions en temps d'execució**: Corresponen a condicions excepcionals internes a l'aplicació, i de les quals l'aplicació tampoc no se'n podrà recuperar. Generalment indiquen errors en la programació, per exemple, utilitzar un objecte que no s'ha creat prèviament.

El control d'excepcions es fa mitjançant la estructura try-catch-finally:

```
try {
      // Codi que volem executar
} catch (TE1 nomVariable) {
     /* Codi que s'executarà si es produeix una
        excepció del tipus TP1. En la variable "nomVariable"
        hi ha tota la informació de la excepció.
     */
} catch (TE2 nomVariable) {
     /* Codi que s'executarà si es produeix una excepció
        del tipus TP2. Igual que en el cas anterior, tindrem
        tota la informació sobre la excepció.
     */
} finally {
      /* El codi que es posa en el bloc "finally", s'executa
      sempre, tant si s'ha produït una excepció com si tot ha
      anat bé.
      */
}
```

Fixeu-vos que podem tenir varis blocs "catch", cadascun per a un determinat tipus d'excepció. Cal tenir en compte que hi ha excepcions més genèriques que altres, i per tant cal posar els blocs "catch" des del que tracta l'excepció més específica fins al que tracta l'excepció més genèrica, en cas contrari sempre s'entrarà al més genèric.

El bloc "finally" sempre s'executa, tant si s'ha arribat al final del bloc "try" correctament com si s'ha entrat en algun dels blocs "catch". Una bona pràctica és posar el codi de neteja (per exemple tancar fitxers) en aquest bloc.

Algunes vegades no ens interessarà tractar una excepció en un cert mètode, sinó que voldrem que el mètode que ens ha cridat tingui coneixement de que s'ha produït aquesta excepció. Et tal cas, en comptes d'utilitzar el bloc "try-catch-finally", simplement podem indicar al Java que aquell mètode "llança" unes certes excepcions.

```
void llegirBiblioteca(String nomFitxer)
                        throws Excepcio1,Excepcio2 {
      .....
}
```

# 4. Referències

- http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html