TEMA 8

ENTRADA, EIXIDA I EXCEPCIONS

Índex de continguts

1 INTRODUCCIÓ

Una part important de tot llenguatge de programació és com s'establirà el contacte amb l'exterior, bé siga mitjançant la consola o a través del sistema de fitxers, per finalment poder actuar amb l'usuari d'una aplicació en execució. Ambdós, consola i sistema de fitxers, poden funcionar tant d'entrada als nostres programes com d'eixida.

Java defineix una abstracció, els streams o fluxos per tractar la comunicació d'informació amb l'exterior.. Per entendre-ho millor, un stream o flux és com un canal de comunicació entre un origen i un destí. Per exemple suposem el cas d'un programa que quan arranque haja de llegir una llista de clients d'un fitxer de text per crear una agenda. En aquest cas l'origen seria el fitxer de text i el destí el programa o aplicació que s'encarrega d'emplenar l'agenda amb el llistat de clients, doncs bé, l'stream en aquest escenari seria el canal de comunicació que s'establiria entre el fitxer de text i el programa per tal que aquest puga llegir la informació.

En aquest tema explicarem com Java tracta mitjançant objectes l'establiment d'aquests canals de comunicació i quines són les operacions que hi podem fer i com s'han d'utilitzar aquestes abstraccions de Java.

D'altra banda Java també ofereix el mecanisme d'excepcions per controlar els errors sobrevinguts. Una excepció és un problema que es produeix en temps d'execució. Java permet controlar aquests problemes i programar amb els try...catch la solució als errors en temps d'execució.

2 ENTRADA I EIXIDA

Com hem dit abans, Java organitza la entrada i eixida mitjançant l'ús d'streams, que són abstraccions en realitat, i aquests streams el que fan es transportar la informació del programa o aplicació a dispositius externs. Aquests dispositius externs poden ser bé fitxers o inclús la consola, que es tractada com un fitxer per Java.

2.1 Flux i arxius.

Un arxiu o fitxer és un conjunt de bytes que són emmagatzemats en un dispositiu. Un arxiu o fitxer es identificat per un nom i la descripció de la carpeta o directori (la ruta) que el conté. Una de les principals finalitats dels arxius és tindre desats en memòria secundària, quan les aplicacions ja han acabat d'executar-se, i que siguen recuperables després.

Un fitxer s'ha de poder llegir, actualitzar, esborrar registres i tornar a guardar-se de nou amb tots els canvis realitzats. Segons el dispositiu físic que els emmagatzeme, els arxius poden ser directes o seqüencials, el primer implica que per poder accedir a qualsevol registre, hem de passar prèviament pels anteriors, per exemple una cinta magnètica, mentre que el segon tipus d'arxiu permet l'accés directe al registre en qüestió sense haver de passar pels anteriors.

En Java un arxiu és una seqüencia de bytes que contenen la informació emmagatzemada. Per poder treballar amb aquestes seqüencies de bytes, Java disposa de classes com són els tipus bàsics (int, double, string..).

D'una altra banda, com ja hem explicat abans, un flux és una abstracció que es refereix a una corrent (stream) de dades entre un origen (també conegut com a font o productor) i una destinació o embornal (consumidor) i la connexió que existeix entre els dos també es coneix com a pipe o tub per on circules les dades.

Quan comença qualsevol execució d'un programa Java, es creen tres objectes flux, canals pels que pot fluir informació d'entrada o eixida; aquests objectes estan definits a la classe System i són:

- Sytem.in; entrada estàndard, permet l'entrada de dades des del teclat.Una excepció és un fallo que es produeix en temps d'execucióÍndex de continguts
- System.out: eixida estàndard, permet la programa imprimir per pantalla.

- System.err: eixida d'errors, permet al programa imprimir errors per pantalla.

2.2 Class File

Per poder identificar d'un fitxer al sistema de fitxers de qualsevol sistema operatiu, necessitem: nom i ruta. Per exemple '/user/data/file.txt'. El fitxer es diu file.txt i la ruta en la qual es troba és '/user/data/'. Aquest identificador de fitxer és el que es passa al constructor de la classe per tal d'obrir un flux.

```
File fitxer = new File (strPath + "prova.txt");
```

En l'exemple anterior hem utilitzat la classe File per tal d'instanciar un stream al fitxer que li hem passat per paràmetre «prova.txt». Compte que s'ha de passar la ruta absoluta d'on es troba el fitxer.

Hi ha alternatives al constructor de la classe File que hem mostrat a l'exemple anterior, es pot construir també amb dos paràmetres: ruta i nom del fitxer o inclús indicar-li la ruta mitjançant un altre objecte File.

Els mètodes de la classe File són:

public boolean exists()

Torna true si el fitxer existeix

---

public boolean canWrite()

Torna true si es pot escriure al fitxer.

---

public boolean canRead()

Torna true si es només de lectura.

---

public boolean isFile()

Torna true si és un fitxer.

---

public boolean isDirectory()

Torna true si és un directori.

---

public boolean isAbsolute()

Torna true si el directori té ruta completa

---

public long lenght()

Torna la mida en bytes del fitxer.

---

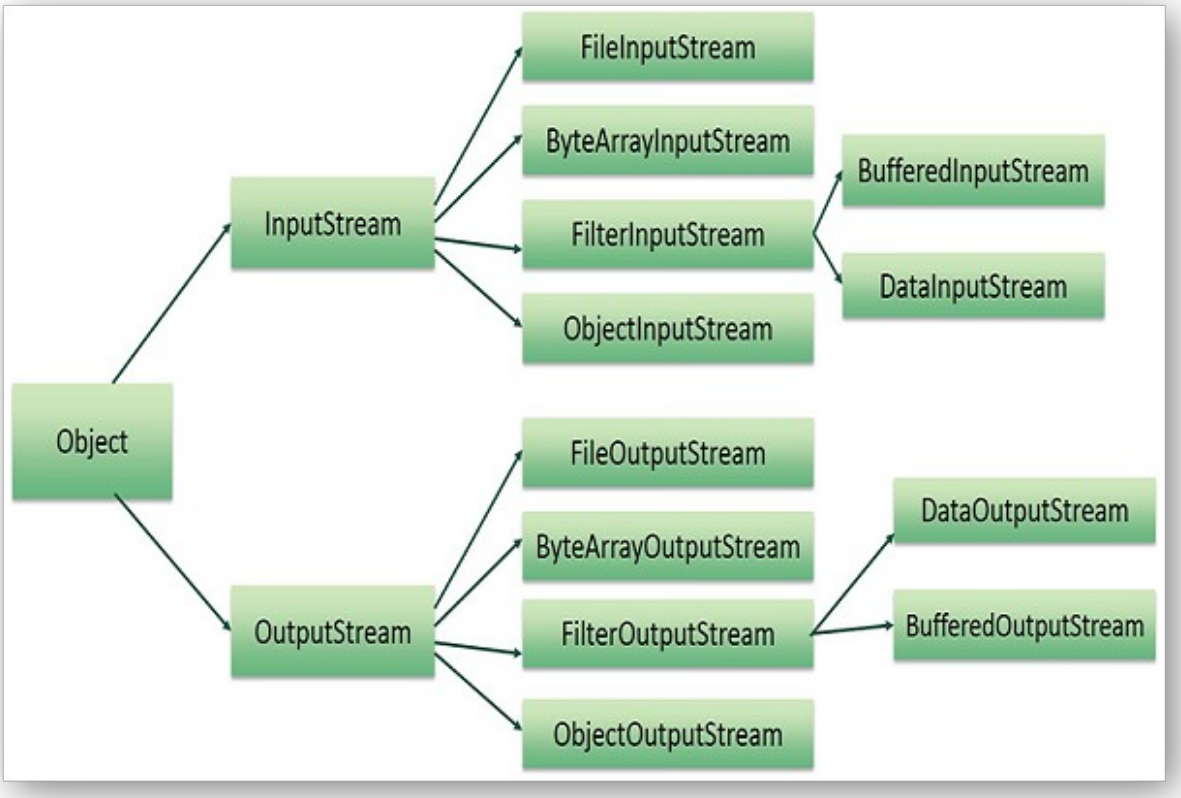
public long lastModified() Torna el timestamp de l'última modificació.
public String getName() Torna una string amb el nom del fitxer.
public String getPath() Torna una string amb el path del fitxer.
public String getAbsolutePath() Torna la ruta absoluta del fitxer.
public boolean setReadOnly() Converteix el fitxer en només lectura.
public boolean delete() Elimina el fitxer o directori (si està buit)
public boolean renameTo(File nou) Canvia el nom pel del fitxer nou.
public boolean mkdir() Crea el directori del fitxer.
public String[] list() Torna un array d'strings dels elements.

2.3 Flux i jerarquia de classes

Com que les possibilitats que tenim en programació respecte de l'ús de fitxers és ampla, podem: llegir de fitxers, escriure en fitxers, llegir per una entrada com un teclat System.in o escriure en una eixida com un monitor System.out, Java declara dues classes que deriven directament de la superclasse «Object» que són: InputStream i OutputStream, ambdues abstractes, és a dir, no podem instanciar cap objecte d'aquestes classes.

InputStream és la base per a totes les classes definies per al flux d'entrada de dades mentre que OutputStream és la classe base per a les classes que gestionen el flux d'eixida.

Bàsicament en la següent imatge podeu les principals classes derivades dels fluxos d'entrada i eixida i la seua jerarquia.



2.4 Arxius de baix nivell: FileInputStream i FileOutputStream

Tot arxiu es pot considerar en sí mateix una seqüència de bytes de baix nivell i després sobre aquesta informació es pot construir informació de mes alt nivell per tal de processar dades més complexes des de tipus bàsic fins objectes.

En altres paraules, si volem obrir un flux de dades a un fitxer o arxiu i volem llegir byte a byte per després decidir nosaltres a nivell de programació que volem fer amb aquests bytes, hauríem d'utilitzar aquestes classes.

```
FileInputStream fen = new FileInputStream(strPath + "prova.txt");
```

La classe FileInputStream s'utilitza per llegir bytes o grups de bytes del fitxer associat. Tots els mètodes tenen visibilitat pública i s'ha de tenir en compte quina és l'excepció que llencen per controlar els possibles errors en temps d'execució.

FileIntpuStream(String nom) throws FileNotFoundException;

Crea un objecte inicialitzat amb el nom d'arxiu que es passa com argument.

FileInputStream(File nombre) throws FileNotFoundException;

Crea un objecte inicialitzat amb l'objecte file que es passa per paràmetre.

---

int read() throws IOException;

Llig un byte del dispositiu. Torna -1 si arriba al final del fitxer.

---

int read(byte[] s) throws IOException;

Llig una seqüència de bytes del flux. -1 si arriba al final del fitxer.

---

int read(byte[] s, int org, int len) throws IOException;

Llig una seqüència de bytes del flux i la desa a 's' començant per 'org' i una mida de 'len'. Torna -1 si arriba al final del fitxer.

---

void close()throws IOException;

Tanca el flux i el fitxer queda lliure per al seu ús posterior.

La classe FileOutputStream seria la complementària a l'anterior. Aquesta, per contra, ens permet escriure bytes o grups de bytes al fitxer que tinga associat el flux.

```
FileOutputStream fout = new FileOutputStream(strPath + "prova.txt");
```

FileOutputStream(String nombre) throws IOException;

Crea un objecte inicialitzat amb el nom d'arxiu que es passa com argument.

FileOutputStream(String nombre, boolean sw) throws IOException;

Crea un objecte inicialitzat amb el nom d'arxiu que es passa per paràmetre. Si sw és true els nous bytes s'afegeixen al final.

FileOutputStream(File nombre) throws IOException;

Crea un objecte inicialitzat amb l'objecte file que es passa per paràmetre.

---

void write(byte a) throws IOException; Escriu el byte 'a' al flux associat.
void write(byte[] s) throws IOException; Escriu l'array de bytes 's' al flux
void write(byte[] s, int org, int len) throws IOException; Llig una seqüència de bytes del flux i la desa a 's' començant per 'org' i una mida de 'len'. Torna -1 si arriba al final del fitxer.
void close()throws IOException; Tanca el flux i el fitxer queda lliure per al seu ús posterior.

Exemple. A continuació obrirem el fitxer «prova.txt» amb un FileInputStream per llegir-lo i desarem el seu contingut a un fitxer d'eixida «resultat.txt» que obirem amb un FileOutputStream. Al fitxer d'eixida, bolcarem el contingut de «prova.txt»

```
try {

    FileInputStream fen = new FileInputStream(strPath + "prova.txt");
    FileOutputStream fout = new FileOutputStream(strPath + "resultat.txt");

    int byteLlegit = fen.read();
    while ( byteLlegit != -1 ) {
        fout.write(byteLlegit);
        byteLlegit = fen.read();
    }

    fen.close();
    fout.close();
}
catch (FileNotFoundException f ) {
    System.out.println("Error Fitxer no trobat");
}
catch (IOException e ) {
    System.out.println("Excepcio IOException");
}
```

Activitat. Basat en el codi anterior i fes que al fitxer resultat, el d'eixida, es copie el que apareix a «prova.txt», el d'entrada però a l'inrevés.

2.5 Arxius de dades

La veritat és que treballar amb bytes quan estem parlant de fitxers o arxius que contenen informació o dades de més nivell pot resultar incomode, ja que entre altres hauríem de saber quants bytes té un caràcter, un enter o un doble.

En Java tenim les classes DataInputStream i DataOutputStream que són subclasses de les classes FilterInputStream i FilterOutputStream

```
FileInputStream gs = new FileInputStream(strPath + "prova.txt" );
DataInputStream ent = new DataInputStream(gs);
```

Bàsicament aquestes classes el que fan és organitzar els bytes en tipus primitius de tal forma que ja podem treballar llegint i escrivint als fluxos: enters, caràcters, reals, cadenes etc..

public DataInputStream(InputStream entrada) throws IOException

Crea un objecte associat a l'stream que es passa per paràmetre

---

public final boolean readBoolean() throws IOException4

Torna el valor de tipus booleà llegit

---

public final byte readByte() throws IOException

Torna el valor de tipus byte llegit

---

public final short readShort() throws IOException

Torna el valor de tipus short llegit

---

public final char readChar() throws IOException

Torna el valor de tipus char llegit

---

public final int readInt() throws IOException

Torna el valor de tipus int llegit.

---

public final long readLong() throws IOException

Torna el valor de tipus long llegit

---

public final float readFloat() throws IOException

Torna el valor de tipus float llegit

---

public final double readDouble() throws IOException

Torna el valor de tipus double llegit

---

public final String readUTF() throws IOException

Torna una cadena que es va escriure en UTF

D'aquesta forma ja no ens hem de preocupar de saber quants bytes de memòria s'han de reservar per un caràcter o un enter. Per exemple si associem

un DataOutputStream a un fitxer i després cridem al mètode writeInt, realment el que s'està fent és escriure 4 bytes d'informació al fitxer de la qual cosa s'estaria encarregant de manera trans11parent a l'usuari.

```
FileOutputStream fn = new FileOutputStream(strPath + "result.txt");
DataOutputStream snb = new DataOutputStream(fn);
```

public DataOutputStream(OutputStream destino) throws IOException

Crea un objecte associat a l'out11putstream que es passa per paràmetre

---

public final void writeBoolean(boolean v) throws IOException

Escriu un boolean

---

public final void writeByte(int v) throws IOException

Escriu un byte

---

public final void writeShort(int v) throws IOException

Escriu un short

---

public final void writeChar(int v) throws IOException

Escriu un char

---

public final void writeChars(String v) throws IOException

Escriu tots els chars que conté la String v

---

public final void writeInt(int v) throws IOException

Escriu un int

---

public final void writeLong(long v) throws IOException

Escriu un long

---

public final void writeFloat(float v) throws IOException

Escriu un float

4public final void writeDouble(double v) throws IOException

Escriu un double

---

public final void writeUTF(String cad) throws IOException

Escriu la cadena cad en format UTF

---

public final int close()throws IOException

Tanca el flux

A continuació podem veure un exemple en el que fem ús del DataOutputStream i DataInputStream. El que fa aquest exemple és escriure a un fitxer d'eixida «pluges.dat» dos valors double aleatoris per després llegir-los amb un DataInputStream i mostrar-los per pantalla.

El codi podria ser el següent:

```
try {

    FileOutputStream fos = new FileOutputStream(strPath + "pluges.dat");
    DataOutputStream dos = new DataOutputStream(fos);

    double info = Math.random()*100;
    dos.writeDouble(info);

    info = Math.random()*100;
    dos.writeDouble(info);

    FileInputStream fis = new FileInputStream(strPath + "pluges.dat");
    DataInputStream dis = new DataInputStream(fis);

    double info2 = dis.readDouble();
    System.out.println("Primer valor: " + info2);
    info2 = dis.readDouble();
    System.out.println("Segon valor: " + info2);

}
catch (FileNotFoundException e) {
    System.out.println("Fitxer no existeix" );
}
catch (IOException e ) {
    System.out.println("Error inesperat IO");
}
```

2.6 Flux PrintStream

La classe PrintStream deriva de FilterOutputStream i el valor afegit d'aquesta classe derivada és que permet afegir el caràcter final de línia als fitxers. Aquests tipus de fluxos són sempre d'eixida i s'associen a un altre flux de bytes de més baix nivell

```
PrintStream ps = new PrintStream (new FileOutputStream("Complex.dat"));
```

public PrintStream(OutputStream destino)

Crea un objecte associat a qualsevol objecte d'eixida per paràmetre

---

public PrintStream(OutputStream destino, boolean flag)

Crea un objecte associat a qualsevol objecte d'eixida per paràmetre. Si flag és true, es produeix un bolcat automàtic al escriure final de línia.

---

public void flush()

Bolca el flux actual

---

public void print(Object obj)

Escriu la representació de l'objecte obj al flux

---

public void print(String cad)

Escriu la cadena al flux

---

public void print(char c)

Escriu un caràcter al flux

---

public void println(Object obj)

Escriu la representació de l'objecte al flux i final de línia

---

public void println(String cad)Crea un fluxe associat amb un altre d'eixida de caracters de tipus Writer.

Escriu la cadena al flux i final de línia

2.7 Arxius de caràcters: Reader i Writer

Reader i Writer són fluxos de Java orientats a caràcters. Amb aquests fluxos podem llegir i escriure caràcters o cadenes de caràcters al dispositius connectats mitjançant aquests fluxos.

Per llegir arxius de caràcters s'utilitzen fluxos derivats de la classe Reader on es declaren o sobreescriuen mètodes per la lectura de caràcters. Els mètodes més importants són:

---

public int read()

Llig un caràcter en forma d'enter. Si arriba al final del fitxer torna -1

---

public int read(char [] b)

Llig una seqüencia de caràcters fins completar l'array b o llegir el final del fitxer. Torna el nombre de caràcters llegits o -1 si arriba al final de l'arxiu.

InputStreamReader

Els fluxos de la classe IntpuStreamReader envolten a un flux de bytes; converteixen la seqüència de bytes en seqüència de caràcters i així ja no ho hem de fer nosaltres; la classe deriva directament de Reader, pel que té disponibles els mètodes read() de la seua classe pare per la lectura de caràcters

```
InputStreamReader ent = new InputStreamReader(System.in);
```

FileReader

Per llegir arxius de text o de caràcters es pot crear un flux del tipus FileReader, aquesta classe deriva d'InputStreamReader, hereta els mètodes read() per a llegir caràcters, el constructor té com entrada una cadena amb el nom de l'arxiu

Per exemple

```
FileReader fr = new FileReader("cartas.dat");
```

En general no resulta eficient llegir directament d'un flux d'aquest tipus, s'utilitzarà un flux BufferedReader.

BufferedReader

La lectura d'arxius de text es realitza amb un flux que emmagatzema els caràcters en un buffer intermedi, aquests no es lligen directament de l'arxiu si no del buffer. D'aquesta manera augmentem l'eficiència a les operacions d'entrada, la classe BufferReader permet crear fluxos de caràcters amb buffer que no és més que una forma d'organitzar el flux bàsic de caràcters del que ve el text perquè al crear un flux BufferReader, aquest s'inicialitza com un flux de caràcters InputStreamReader o qualsevol altre.

El constructor de la classe té un argument de tipus Reader, FileReader o InputStreamReader, el flux creat disposa d'un buffer de grandària suficient, el qual es pot especificar en el constructor amb un segon argument encara que no fa falta. Exemples de fluxos amb buffer;

Writer

Els arxius de text són arxius de caràcters, es poden crear fluxos de bytes o de caràcters derivats de la classe abstracta Writer, la qual defineix mètodes write() per escriure arrays de caràcters o cadenes, d'aquesta classe es deriva OutputStreamWriter que permet escriure caràcters en un flux de bytes al qual s'associa la creació de l'objecte o flux.

```
OutputStreamWriter ot = new OutputStreamWriter(new FileOutputStream(archivo));
```

No es freqüent utilitzar directament fluxos OutputStreamWriter, encara que resulta interessant perquè la classe FileWriter és una extensió d'ella. Dissenyada per escriure en un arxiu de caràcters, els fluxos d'aquest tipus escriuen caràcters amb el mètode write() a l'arxiu al que s'associa el flux quan es crea l'objecte.

```
FileWriter nr = new FileWriter("cartas.dat");
nr.write("Frase qualsevol a escriure");
```

PrintWriter

Els fluxos més utilitzats en l'eixida de caràcters són de tipus PrintWriter, aquesta classe declara constructors per associar un flux PrintWriter amb qualsevol altre de tipus Writer o bé OutputStream.

---

public PrintWriter(OutputStream destí)

Crea un flux associat amb un altre d'eixida a nivell de byte.

---

public PrintWriter(Writer destino)

Crea un flux associat amb un altre d'eixida de caràcters de tipus Writer.

La importància d'aquesta classe radica en que defineix mètodes print() i println() per cadascun dels tipus de dades simples, per String i per Object; la diferència entre els mètodes print() i println() està en que el segon afegeix els caràcters de final de línia a continuació dels escrits per l'argument.

---

public void print(Object obj)

Escriu la representació de l'objecte obj al flux

---

public void print(String cad)

Escriu la cadena al flux

---

public void print(char c) Escriu el caràcter c al flux.
public void println(Object obj) Escriu la representació de l'objecte obj al flux i final de línia
public void println(String cad) Escriu la cadena al flux i el final de línia
public void println(char c) Escriu el caràcter c al flux i final de línia.

2.8 Arxius d'objectes

Per tal que un objecte continue existint una vegada ja hem finalitzat l'execució d'un programa o aplicació l'hem de desar a un arxiu d'objectes. Per poder aconseguir açò, utilitzarem les classes ObjectInputStream i ObjectOutputStream

CLASSE D'OBJECTE PERSISTENT

La declaració de la classe els objectes de la qual van a ser persistents, han de implementar la interfície «serializable» del paquet java.io, la qual és buida, no declara mètodes, simplement indica a la màquina virtual de Java que les instàncies d'aquestes classes podran gravar-se en un fitxer.

```
class racional implements serializable { ... }
```

FLUX OBJECTOUTPUTSTREAM

Els fluxos de la classe ObjectOutputStream s'utilitzen per gravar objectes persistents. El mètode writeObject() escriu qualsevol objecte d'una classe serializable en el flux de bytes associat. Pot llençar excepcions del tipus IOException que és necessari processar.

```
public void writeObject(Object obj) throws IOException;
```

El constructor de la classe espera un argument de tipus OutputStream, que és la base dels fluxos d'eixida a nivell de bytes, per tant, per crear aquest tipus de fluxos primer es crea un d'eixida a nivell de bytes associat a un arxiu extern i a continuació, es passa com argument al constructor de ObjectOutputStream, per exemple:

```
FileOutputStream bn = new FileOutputStream("datosRac.dat");
ObjectOutputStream fobj = new ObjectOutputStream(bn);
```

A continuació es pot escriure qualsevol objecte al flux:

```
Persona juan = new Persona("Juan", "Martin", "20392020S");
fobj.writeObject(juan);
String str = new String("Cadena de favores");
fpbj.writeObj(str);
```

FLUX OBJECTINPUTSTREAM

El objectes guardats en arxius amb fluxos de la classe ObjectOutputStream es recuperen i lligen amb fluxos d'entrada del tipus ObjectInputStream, aquesta classe és una extensió d'InputStream, a més a més, implementa la interfície DataInput, per això disposa de diversos mètodes d'entrada (read) per cadascun dels tipus de dades com readInt() o altres. El mètode més important definit per la classe ObjectInputStream és readObject(), el qual llig un objecte del flux d'entrada, és a dir, de l'arxiu associat al flux de baix nivell; l'objecte llegit es va escriure en el seu moment amb el mètode writeObject()

```
public Object readObject() throws IOException;
```

El constructor de fluxos ObjectInputStreeam té com entrada un altre flux de baix nivell de qualsevol tipus derivat d'InputStream, per exemple: FileInputStream associat amb l'arxiu d'objectes. A continuació es crea un flux d'entrada per llegir els objectes de l'arxiu «archivoObjects.dat»

```
ObjectInputStream obj = new ObjectInputStream(new FileInputStream("archivoObjets.dat "));
```

El constructor llença una excepció si, per exemple, l'arxiu no existeix, aquella és del tipus «ClassNotFoundException» o «IOException», és necessari poder capturar aquestes excepcions

3 EXCEPCIONS

Un problema important en el desenvolupament del programari és la gestió de les errades o errors. No importa quan bé estiga planificat aquest desenvolupament i quan eficient siga l'equip desenvolupador. És una màxima en informàtica el fet que sempre apareixeran problemes. Aquests problemes inesperats són problemes o errors que apareixen en temps d'execució, per exemple: esgotament de memòria o recursos, errors en els intervals dels bucles, divisions per zero, arxius no existents, etc..

Les excepcions són el mecanisme previst per Java per tractar aquests problemes sobrevinguts. Amb les excepcions, Java dona la possibilitat al programador salvar aquest tipus de situacions de manera controlada sense que el programa o aplicació es tanque, bloquege o deixe de respondre.

3.1 Condicions d'errors en programes

Programar, escriure codi, mètodes, classes, etc.. que siga eficient, eficaç i lliure d'errades és altament complicat. El control d'errades és una part tant important en el disseny d'aplicacions que a totes les diferents metodologia de cicle de vida software, hi ha una fase que es dedica única i exclusivament a detectar aquestes errades i solucionar-les i així i tot és impossible produir programari totalment lliure d'errades.

3.2 Tractament dels codis d'error

Java inclou un mecanisme de gestió d'excepcions per poder intentat cobrir possibles errades en els programes en temps d'execució. Aquest mecanisme consisteix en capturar (catch) els errors quan es produeixen en temps d'execució.

Quan es produeix un fallo en temps d'execució es podrien fer tres coses: parar l'execució de sobte, continuar executant esperant que no passe res o establir una senyal d'error que el mateix codi amb altres sentències s'encarregue de gestionar.

Aquestes senyals que envia (throw) el programa quan es produeix alguna errada en Java s'anomenen excepcions i el bloc del codi on es posen les sentències per tal d'intentar solucionar-lo en temps d'execució és el bloc catch.

3.3 Excepcions en Java

Una excepció és un fallo que es produeix en temps d'execució. Si hem escrit bé el codi per tal de gestionar aquest fallo, el flux d'execució passa al codi destinat a gestionar les excepcions, en cas contrari el programa acaba la seua execució.

Aquestes excepcions es poden produir per exemple si intenten accedir a un element d'un array fora dels seus límits, si intentem obrir un fitxer que no existeix o està bloquejat, si dividim per zero, etc..

El model de gestió d'excepcions a Java afegeix cinc noves paraules reservades: try, throw, throws, catch i finally.

- try: bloc on es detectaran les excepcions
- catch: captura les excepcions que es produiexen al bloc try
- throw: expressió per llençar excepcions
- throws: indica les excepcions que pot llençar un mètode.
- finally: bloc opcional situat després del try catch

Els pasos del model de control d'excepcions en Java són:

1. El bloc try conté les instruccions o part del codi susceptible de provocar algun error.
2. Si en temps d'execució es produeix algun error dins de les sentències que es troben dins del bloc try, es llença una excepció.
3. S'executa el codi que es troba dins del bloc catch associat a la excepció que s'ha llençat al punt anterior.
4. S'executa el bloc finally (opcional)

Exemple. Suposem una classe persona que incorporarem a la nostra aplicació. Aquesta classe té un constructor al qual se li passa el DNI, imaginem ara que de la nostra aplicació instanciem un objecte de la classe persona a la qual l direm Joan però quan cridem al constructor de la classe no li passem un DNI, aquest «mal ús» de la classe Persona, llençaria una excepció que hauria de capturar la nostra aplicació al bloc catch i és ací, en aquest bloc d'instruccions, en el que s'hauria de corregir la situació o avisar a l'usuari de l'aplicació del que ha passat.

```java
public static void escolta () throws Exception {

    // Codi del mètode escolta
    // ...
    throw new Exception();
}


public static void main (String [] args) {

    try {
        escolta();
    }
    catch ( Exception e ) {
        // Codi que s'ha d'executar quan es produeix l'excepció
    }
}
```

Al codi anterior hem crear un mètode escolta que l'únic que fa és llençar una excepció. Quan al mètode main cridem a la funció escolta, si volem fer ús del sistema d'excepcions de Java, l'haurem de posar dins del bloc try. Just després d'acabar el conjunt d'instruccions del bloc try posem un catch indicant-li quina és la classe d'excepció que hauriem de capturar, al nostre cas Exception, i dins del bloc catch posariem el codi que s'hauria d'executar en cas de produir-se eixe error. Normalment es solen posar missatges d'error que alerten l'usuari de l'aplicació.

A continuació explicarem amb més detall què és el que fan cadascun dels blocs del mecanisme d'excepcions que ha dissenyat Java.

3.4 Bloc try

Dins del bloc try s'han de posar aquelles sentències que poden llençar alguna excepció entre claudàtors, després del claudàtor de tancament s'ha de posar el bloc catch o controlador d'excepcions almenys un bloc catch. Si l'excepció que es produeix coincideix amb algun dels paràmetres dels blocs catch que hi han, s'executen les sentències que es troben a dintre.

Els blocs try, de la mateixa forma que amb totes les sentències de programació estructurada, es poden niar.

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

    try {
        escolta();

        try {
            tornaAEscolar();
        }
        catch ( Exception e ) {
            // Codi d'excepció
        }
    }
    catch ( Exception e ) {
        // Codi que s'ha d'executar quan es produeix l'excepció
    }
    catch (FileNotFoundException fe) {
        // Codi del segon catch
    }
}
```

A l'exemple anterior és pot observar un codi amb dos blocs try niats on el primer bloc try té dos blocs catch per controlar varies excepcions

3.5 Bloc throw

La sentència throw llença (raise) una excepció. Les excepcions en Java són objectes, per tant, una excepció és una instància d'una classe que deriva de la classe Exception

A l'exemple anterior com es pot observar es capturen dos tipus d'excepcions diferents, una generica Exception i una altra FileNotFoundException. Aquesta última es produiex normalment quan volem obrir un fitxer, amb la classe File per exemple, i aquest no existeix.

3.6 Bloc catch

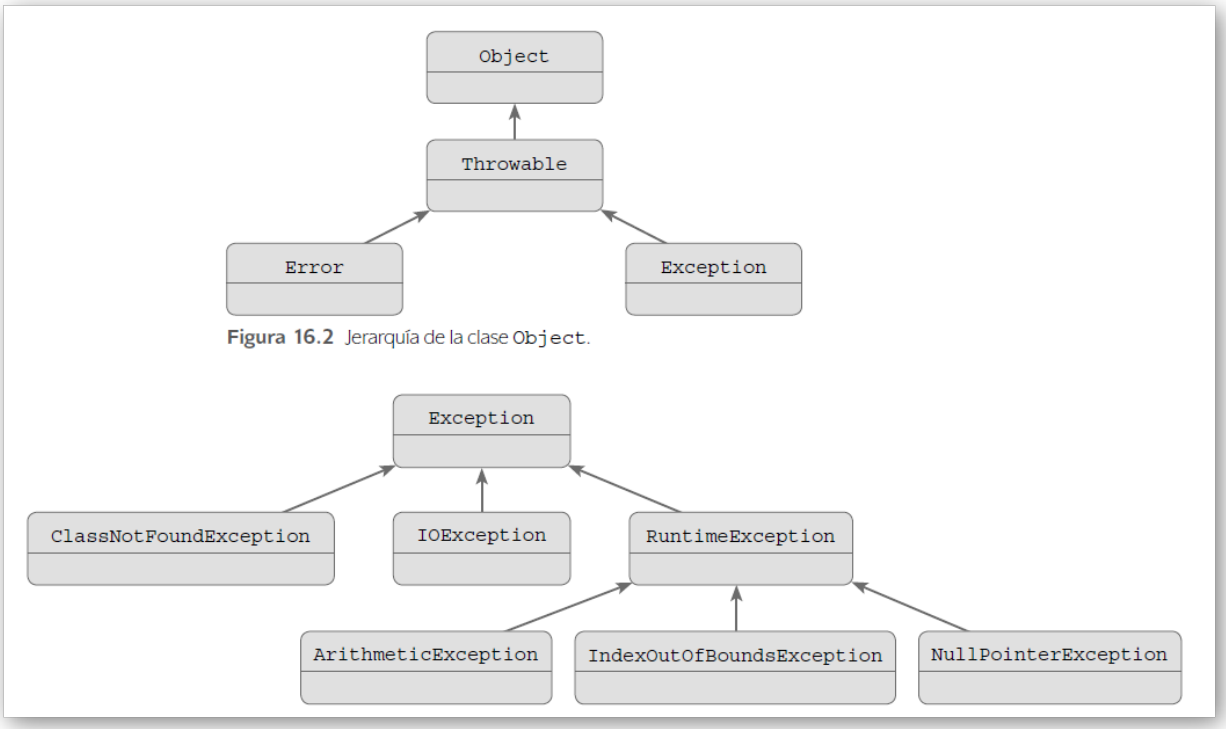
El bloc catch o de captura d'excepcions. Quan una excepció es llença des d'un bloc try, si el tipus d'excepció que s'ha llençat coincideix amb el tipus d'excepció que s'ha passat per paràmetre al catch, s'executarien les sentències que hi han dins del bloc catch. Normalment es sol posar el codi necessari per poder eixir de la situació en la que ens trobem quan s'alça una excepció.

3.7 Clàusula finally

El bloc finally en un try..catch, és el bloc d'instruccions que s'executarà sempre, hi haja excepcions o no, al final de tot el nostre codi.

Normalment es solen posar sentències per tal d'alliberar tot allò que pugam haver creat dins del bloc try, és a dir, tancar fluxos, fitxers, connexions a bases de dades etc..amb la finalitat d'alliberar memòria.

3.8 Classes d'excepcions definides a Java



3.9 Noves classes d'excepcions

Java permet la creació d'excepcions noves que deriven de la classe base Exception