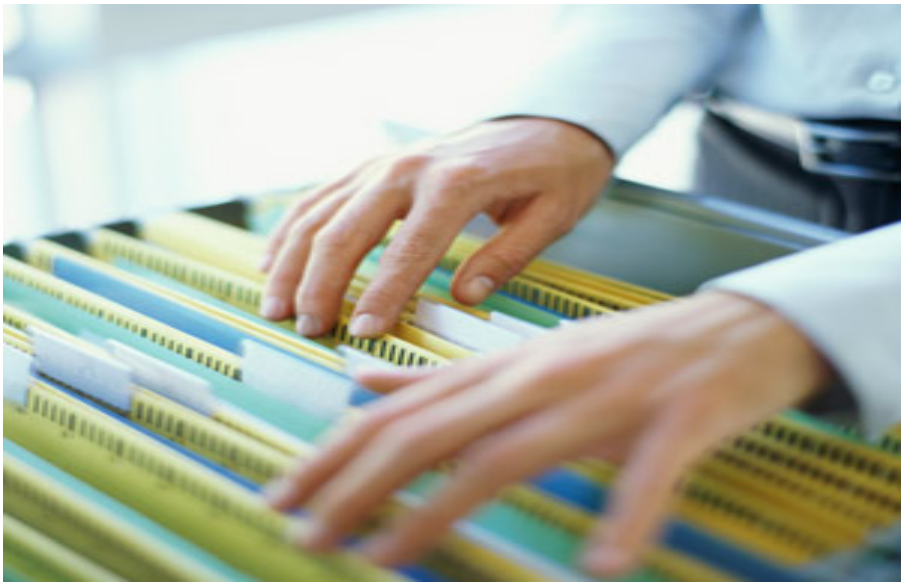
U.T. 5 - ENTRADA I EIXIDA DE DADES



1 - PERSISTÈNCIA DE DADES

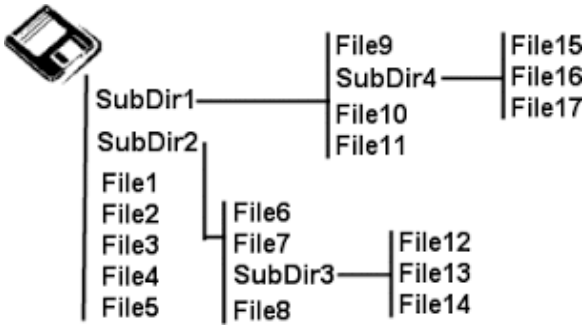
Un arxiu o fitxer és un recull de dades, generalment homogenis, emmagatzemats en un suport físic de la màquina, fix o extraïble. Per dades homogènies entenem aquells en què emmagatzemem col·leccions de dades del mateix tipus (similar als arrays / vectors). Cada element emmagatzemat (per exemple, dades de llibres d'una biblioteca) es denomina **registre** (cada llibre), que es compon de camps (títol, autor, ISBN, etc.). El suport físic més habitual sol ser el disc dur, però també ho poden ser unitats externes d'emmagatzematge, disquets, etc ...

Fins ara, en unitats anteriors, hem treballat en tot moment amb estructures de dades guardades en memòria RAM. La memòria RAM ofereix com a principal avantatge el ràpid accés a les seves dades, amb temps sempre inferiors als nanosegons, però presenta dues importants desavantatges:

- les dades emmagatzemades es perden al acabar el programa
- capacitat d'emmagatzematge limitada

Aquests dos inconvenients són els que podrem evitar fent ús de memòries secundàries com el disc dur. Això ens permetrà guardar les dades, acabar l'aplicació i recuperar-les en posteriors execucions. L'inconvenient serà que els temps d'accés seran molt més llargs, de l'ordre de milisegons.

2 - CLASSE FILE



El treball amb arxius en cert aspecte no és tan senzill, ja que cal verificar i tenir molts elements baix el nostre control com són els permisos, que l'arxiu existisca, evitar sobreescriure i perdre les dades ja guardades, etc. Per controlar tots aquests aspectes, Java ens ofereix la classe **File**; amb ella podem tractar a l'arxiu com un objecte i beneficiar-nos de les eines i funcionalitats que té la classe.

File inclou en una col·lecció de mètodes i constructors que ens faciliten la creació d'arxius en el sistema, així com també l'accés a les diferents propietats dels mateixos, com ara la ruta absoluta de l'arxiu, els seus permisos, etc.

Per ser multiplataforma Java ens ajuda a lluitar amb aspectes específics de cada sistema operatiu, com els separadors de ruta, diferents a Windows i en sistemes basats en Unix, com Linux. Incidint en el punt de les rutes, no és una pràctica recomanable utilitzar rutes absolutes, ja que això limita molt la portabilitat del nostre programa, per exemple una ruta: "c:\llibre\tutorial.java" ens va a causar problemes si executem el nostre programa en un entorn diferent a Windows (pel separador \ per a Windows, que difereix de / dels sistemes Linux, i també és el separador de Java per defecte).

La forma recomanada és deixar que la classe *File* s'encarregue d'això per nosaltres:

```
File f = new File("imatges/imatge.jpg");
```

Aquesta línia fa que la nostra aplicació busque un directori anomenat "imatges" dins del directori on està el nostre programa i cree l'objecte associat al fitxer corresponent. El fitxer no és necessari que existisca prèviament.

Anem ara a fer un xicotet programa d'exemple on podrem utilitzar diversos mètodes d'aquesta classe per obtenir detalls d'un arxiu:

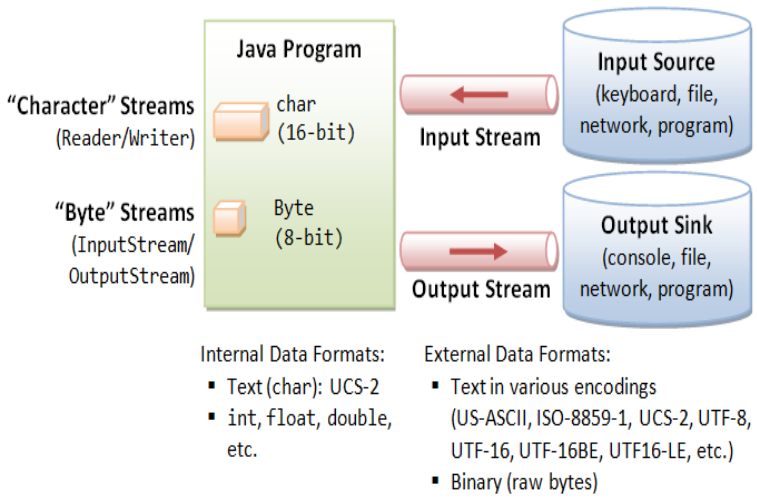
```
public class TestFileClass {
    public static void main (String [] args) {
            java.io.File f = new java.io.File("imatges/imatge.jpg");
            System.out.println( "Existeix: " + f.exists ());
            System.out.println( "Té una mida de " + f.length () + " bytes");
            System.out.println( "Pot ser llegit? " + f.canRead ());
            System.out.println( "Pot ser escrit? " + f.canWrite ());
            System.out.println( "És un directori? " + f.isDirectory ());
            System.out.println( "És un arxiu? " + f.isFile ());
            System.out.println( "És absolut? " + f.isAbsolute ());
            System.out.println( "Està ocult? " + f.isHidden ());
            System.out.println( "La ruta absoluta és " + f.getAbsolutePath ());
            System.out.println( "Ultima modificació: " + new Date (f.lastModified ()));
    }
}
```

3 - TIPUS DE FITXERS

Els fitxers poden ser classificats per diferents criteris. Pel que es requereix a la forma d'escriptura, distingim entre fitxers **binaris** i **de text**. En els primers guardem les dades sense transformacions, és a dir, un sencer amb els seus 4 bytes (1 bit per al signe, 31 per la magnitud), mentre que en els de text guardem les dades com seqüències de caràcters ASCII (la qual cosa també pot comprendre valors numèrics, però cada dígit amb un caràcter ASCII).

Un altre criteri de classificació pot ser atenent a la seva forma d'accés a les dades o organització. Des d'aquest punt de vista tenim seqüencials, d'accés directe, indexats i altres formes híbrides (combinació de les anteriors).

- Organització **seqüencial**: registres emmagatzemats consecutivament en memòria segons l'ordre lògic en què s'han anat inserint.
- Organització **d'accés directe**: l'ordre físic d'emmagatzematge en memòria no ha de coincidir amb l'ordre en què han estat inserits. Hi ha una relació o fórmula per conèixer la situació de cada registre.
- Organització **indexada**. Inclouen realment dos fitxers: un fitxer principal amb les dades pròpiament dites, i un fitxer d'índex que conté la posició de cadascun dels registres en el fitxer de dades (similar als índexs en els llibres).



Al escollir el tipus de fitxer hem de tenir en compte que el suport físic on residiran les dades condicionarà la decisió. Per exemple, un suport de cinta magnètica només suportarà l'organització seqüencial.

Dels tres tipus esmentats ens centrarem primerament en els **seqüencials**. Com a avantatges d'aquest tipus de fitxers destacarem dos:

- aprofitament màxim de l'espai en la unitat d'emmagatzematge. Les dades es disposen linealment, en seqüència com el seu propi nom indica, igual com passava amb els arrays
- els algorismes per a lectura, escriptura i recerca resulten molt senzills, conseqüència derivada de la linealitat comentada.

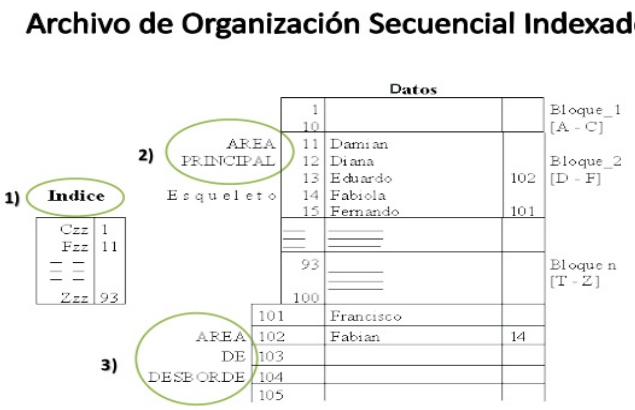
L'inconvenient, fonamental per a desaconsellar-los en grans volums de dades, és el fet de haver de recorrer totes les dades anteriors per a arribar a la dada buscada.

Reader i *Writer* són les classes base de la jerarquia per als fluxos de caràcters, és a dir, els fitxers de text. Per contra, per a llegir o escriure dades binàries utilitzarem una altra jerarquia de classes en les que les classes base són *InputStream* i *OutputStream*. Totes elles les trobem al paquet **java.io**.

El conjunt d'aquestes classes ens permetran realitzar operacions d'entrada, de eixida o ambdues. L'ús més freqüent d'elles pot ser operar amb fitxers, però realment les entrades o eixides a dispositius (com teclat, impressora, etc.) es realitzen amb les mateixes classes i els mateixos mètodes.

Observaràs, a més, que, al treballar amb aquestes classes, el compilador amb freqüència ens advertirà que aquestes operacions d'entrada i eixida poden provocar diferents errors d'execució o excepcions, la més comuna d'elles serà la *IOException*. Hauràs d'incloure les línies indicades pel mateix compilador dins d'un bloc *try {...}* seguit del corresponent bloc *catch (TipusExcepcio e) {...}*.

Si el compilador t'indica diferents excepcions a gestionar, hauràs de situar-les en l'ordre de més particular a més general. Per exemple, el *catch* corresponent a *FileNotFoundException* hauria d'anar abans del d'*IOException*, ja que la primera excepció hereta de la segona (pots comprovar-ho en la documentació). Si invertirem l'ordre no s'executaria mai el codi del segon *catch*, perquè una excepció *FileNotFoundException* també és una *IOException* (característica "és un" de l'herència).



Altra operació necessària, al acabar de treballar amb aquestes classes, és tancar el flux de dades. Això es fa amb el mètode *close()* que implementa l'interfície *Closeable*, i permet lliurar recursos de memòria del sistema operatiu així com assegurar la integritat de les dades. Com veurem posteriorment podrem automatitzar la crida a *close()*.

4 - FITXERS DE TEXT

Per començar a treballar amb fitxers de text, contràriament al que acabem de dir, anem a començar utilitzant la classe **FileInputStream**. Del seu nom pots deduir que es tracta d'una classe utilitzada per a entrada de dades, és a dir, lectura i en binari (realment els fitxers de text són un tipus particular de binaris, que ho serien tots). Consulta la seva documentació. Observa en ella el mètode:

int read()

Llegeix i retorna, com a sencer, un byte de dades des d'aquest flux d'entrada.

Si coneixes el comandament *cat* del sistema operatiu, se't proposa fer la teva pròpia versió. El programa hauria d'acceptar en l'execució un paràmetre, que continga el nom o la ruta d'un fitxer que ja existisca, i bolcar el seu contingut a pantalla. Prèviament, hauràs de crear una instància d'aquesta classe, associada al fitxer indicat amb algun dels seus constructors:

FileInputStream(File file): crea un objecte *FileInputStream* partint d'un objecte *File* prèviament creat i associat a un determinat fitxer.

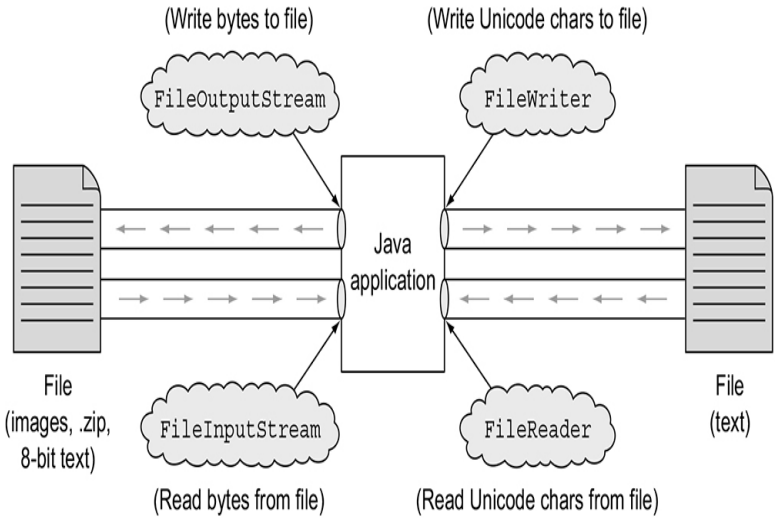
FileInputStream(String name): crea un objecte *FileInputStream* a partir d'un objecte *String* amb el seu nom o ruta en el sistema de fitxers.

Una vegada sàpigues utilitzar aquesta classe no et suposarà dificultat aprendre la corresponent **FileOutputStream**. Del seu nom pots deduir que es tracta d'una classe utilitzada per a eixida de dades, és a dir, escriptura. Si consultes la seva documentació podràs veure que és pràcticament coincident amb l'anterior, però canviant el sentit del flux de dades. En aquest cas pots veure:

void write(int b)

Escriu en aquest fitxer o flux de eixida, el byte passat com a paràmetre.

Observa igualment els seus constructors. Pots veure que el primer permet construir l'objecte a partir d'un objecte *File* prèviament creat, i que també tens un altre que et permet passar el nom o ruta d'el fitxer com String. A més, tens altres constructors que afegeixen un segon paràmetre de tipus booleà anomenat *append* (afegir). Aquest paràmetre, amb el valor *true*, permet mantenir el contingut del flux prèviament existent i afegir a continuació el que escrivim, sense perdre el contingut previ. Dit d'una altra manera, sense aquest booleà a *true*, el comportament per defecte d'aquestes classes és que crea sempre des de zero el fitxer, la qual cosa suposa que si prèviament existís perdríem el seu contingut anterior.



Sabent això ja pots intentar fer una altra versió pròpia d'algun dels comandaments fonamentals de la línia d'ordres: l'ordre *cp* per copiar el contingut d'un fitxer en un altre. Recorda que la seva sintaxi és "cp origen destí". Pots fer que el programa es comporte com el mateix *cp* del sistema operatiu, admetent igualment dos paràmetres: el primer l'origen i el segon el destí de la còpia. La idea serà llegir en un bucle cada caràcter del fitxer d'origen, fent ús de *FileInputStream* i escriure-ho en el fitxer de destinació, amb *FileOutputStream*. El bucle farà tantes passades com caràcters incloga el fitxer (coincident amb la seva grandària, mostrat amb un *"ls -l"*).

Com podràs imaginar, els programes anteriors, fets treballant amb el text caràcter a caràcter, no són la manera més eficient de realitzar aquestes tasques. Una optimització primera i fonamental ha de ser la de poder treballar amb tot un text, amb la seva total extensió, conjuntament. Per a això anem a veure una altra classe que ja és coneguda: *BufferedReader*.

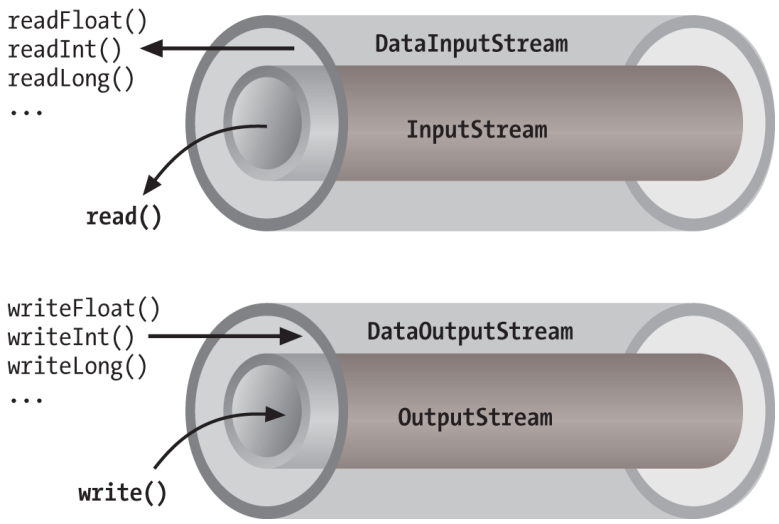
Fins ara hem utilitzat **BufferedReader** per a lectura (entrada de dades) des de teclat. Si mirem, a la seva documentació, els seus constructors podrem veure que podem instanciar objectes d'aquesta classe associats a altres dispositius diferents de teclat (*System.in*), com pot ser un fitxer de text. Observa així mateix que inclou un mètode que ens permetrà llegir tota una línia, sense fer-ho caràcter a caràcter. Sí, efectivament, és el mateix mètode *readLine()*. Se't proposa repetir el programa que bolcava el contingut d'un fitxer de text per pantalla.

Amb ajuda de la documentació també, intentarem fer aquest altre programa: aquell que demane línies de text, fins a acabar amb una cadena buida, i les escriga a fitxer utilitzant la classe **FileWriter**.

5 - FITXERS BINARIS

Com hem vist amb anterioritat, els fitxers binaris són aquells que guarden les dades en representació binària, no textual. En general, qualsevol fitxer que no siga de text serà binari, fins i tot quan part del seu contingut siga textual també.

Per escriure i llegir en binari anem a partir de les classes **DataOutputStream** i **DataInputStream**, i els seus mètodes *writeBoolean*, *writeInt*, *writeDouble*, etcètera, així com *readBoolean*, *readInt*, *readDouble*, etcètera.



Es proposa un exerci inicial per a l'alumne: programa que demane valors numèrics (fins a acabar amb un zero) i els escriga a fitxer.

L'ús més habitual d'aquestes classes serà per guardar, no simples números, sinó registres. Qualsevol programa que guarde registres (escrivint individualment els diferents camps amb els mètodes *writeInt*, *writeDouble*, *writeChars*, *writeUTF* ...) farà ús d'aquestes classes.

Els mètodes *writeChars* i *writeUTF* són mètodes utilitzats per escriure dades de tipus cadena (String) en un flux d'eixida, com pot ser un fitxer. La diferència principal entre ells es troba en el format en què s'escriuen les dades i la forma en què es llegeixen posteriorment.

El mètode *writeChars* escriu dades de tipus cadena en forma de caràcters Unicode. Aquest mètode escriu primer el nombre de caràcters de la cadena (2 bytes) i, a continuació, escriu cada caràcter individual com a valor Unicode (2 bytes per caràcter) en el flux. Aquesta representació de caràcters Unicode pot ser útil en situacions en què interesse que tinga una longitud prevista o fixa.

El mètode *writeUTF* també escriu dades de tipus cadena en un flux d'eixida, però utilitza un format diferent. Aquest mètode escriu primer el nombre de bytes de la cadena (2 bytes) i, a continuació, escriu la seqüència de bytes de la cadena en el flux. A diferència de *writeChars*, writeUTF utilitza una representació de caràcters amb longitud variable, on cada caràcter pot ocupar entre 1 i 3 bytes, depenent del seu valor Unicode. Això permet representar caràcters en diferents idiomes i escriure dades de cadena amb un menor consum de memòria.

A l'hora de llegir les dades des del flux d'eixida, és important utilitzar els mètodes de lectura corresponents (*readChars* o *readUTF*) per interpretar correctament el format de les dades escrites.

Com a exemple de lectura i escriptura de registres, anem a veure un programa que escriga registres en fitxer (poden ser dades d'alumnes, llibres, pel·lícules, discs, etc.) i, posteriorment, mostre el contingut emmagatzemat.

```
import java.io.*;
import java.util.Scanner;
// per a cada empleat anem a registrar identificador (int) ,nom i salari
public class gestioEmpleats
{
    public static void main(String[] args)
    {
        Scanner ent = new Scanner(System.in); int opc;
        do{
            System.out.println("\n1.Alta
                \n2.Consultar empleat
                \n3.Llistar tots els empleats
                \n4.Baixa
```

```
                \n5.Eixir
                \n\tIntroduix opció:");
            opc = ent.nextInt();
            switch(opc)
            {
                case 1: alta();break;
                case 2: System.out.println("Introdueix id:");
                    int n=ent.nextInt();
                    consulta(n);break;
                case 3: llista();break;
                case 4:System.out.println("Introdueix id:");
                    n=ent.nextInt();
                    baixa(n);break;
                case 5: break;
                default: System.out.println("Opció incorrecta");
            }
        } while(opc != 5);
    }

    public static void alta()
    {
        int cod;
        String nom;
        double salari;
        int id;
        try(FileOutputStream fos = new FileOutputStream("empleats.dat",true);
            DataOutputStream dos= new DataOutputStream(fos);
        )
        {
            Scanner ent = new Scanner(System.in);
            System.out.println("Escriu l'identificador enter de l'empleat: ");
            cod=ent.nextInt();
            ent.nextLine();     // per a evitar el problema del buffer
            System.out.println("Escriu el seu nom: ");
            nom= ent.nextLine();
            System.out.println("Escriu el seu salari:");
            salari=ent.nextDouble();
            dos.writeInt(cod);dos.writeUTF(nom);dos.writeDouble(salari);
        }
        catch(IOException e)
        {
            System.err.println(e.getMessage());
        }
    }

    public static void consulta(int codi)
    {
        int id;String nom; double salari;
        try(FileInputStream fis = new FileInputStream("empleats.dat");
            DataInputStream dis = new DataInputStream(fis);)
        {
            id = dis.readInt();
            nom=dis.readUTF();
            salari=dis.readDouble();
            while( id != codi)
            {
```

```
                id=dis.readInt();
                nom=dis.readUTF();
                salari=dis.readDouble();
            }
            if(id == codi)
                System.out.println("\nNom: "+nom+"\n"+"\nSalari: "+salari);

        }
        catch(IOException e)
        {
            System.err.println(e.getMessage());
        }

    }

    public static void baixa(int cod)
    {
        int id;String nom; double salari;
        File f1 = new File("empleats.dat");
        File f2 = new File("empleats2.dat");
        try(FileInputStream fis = new FileInputStream("empleats.dat");
            DataInputStream dis = new DataInputStream(fis);
            FileOutputStream fos = new FileOutputStream("empleats2.dat");
            DataOutputStream dos= new DataOutputStream(fos))
        {
            id=dis.readInt();
            nom=dis.readUTF();
            salari=dis.readDouble();
            while(true)
            {
                if(id != cod)
                {
                    dos.writeInt(id);
                    dos.writeUTF(nom);
                    dos.writeDouble(salari);
                }
                id=dis.readInt();
                nom=dis.readUTF();
                salari=dis.readInt();
            }
        }
        catch(IOException e)
        {
            System.err.println(e.getMessage());
        }
        f2.renameTo(f1);
    }

    public static void llista()
    {
        int cod;String nom; double salari;
        try(FileInputStream fis = new FileInputStream("empleats.dat");
            DataInputStream dis = new DataInputStream(fis);)
        {

            while(true)
```

```
            {
                cod=dis.readInt();
                nom=dis.readUTF();
                salari=dis.readDouble();
                System.out.println("Identificador: "+cod+"\nNom "+nom+"\n
                        Salari:"+salari+"\n");
            }
        }
        catch(EOFException e)
        {
            //System.err.println(e.getMessage());
        }
        catch(IOException e)
        {
            System.err.println(e.getMessage());
        }
    }
}
```

Observa com en la última funció *llista()*, donat que *readInt()* llança una EOFException quan s'arriba al final del fitxer, optem per no fer res en el seu corresponent *catch*. EOF significa "End Of File", final del fitxer. Es tracta per tant d'una excepció prevista, que sempre s'ha d'acabar produïnt.

6 - SERIALITZACIÓ

Extret de la documentació (interfície *Serializable*):

La "serialitzabilitat" d'una classe s'habilita mitjançant la implementació de la interfície *java.io.Serializable*. Les instàncies de les classes que no la implementen no podran serialitzar o deserialitzar el seu estat. Qualsevol subtipus d'una classe serialitzada ho és també. La mateixa interfície no té cap mètode ni atribut, s'utilitza únicament per identificar la semàntica de ser serialitzada.

Això suposa que per a què un programa Java puga convertir un objecte en una seqüència de bytes que puga després ser recuperada, l'objecte necessita ser serialitzat. Al poder convertir l'objecte a bytes, aquest objecte es pot enviar a través de la xarxa, guardar-lo en un fitxer, i després reconstruir-lo a l'altra banda de la xarxa, tornar a llegir-lo, etc.

Perquè els objectes d'una classe siguen serialitzats és tan senzill com implementar la interfície *Serializable*, sense haver de definir cap mètode. Per exemple:

```
public class Dades implements Serializable
{
    private int a;
    private String b;
    private char c;
}
```

Si dins de la classe hi han atributs que són altres classes, aquests al mateix temps també han de ser serialitzables. Amb molts tipus propis del llenguatge (String, Integer, etc.) no hi ha problema perquè ho són. Si posem com a **atributs** les nostres pròpies classes, aquestes **també han d'implementar Serializable**. Per exemple:

```
/* Classe serialitzada: implementa Serializable i els seus atributs ho són*/
public class DadaMajor implements Serializable
{
    private int d;
```

```
    private Integer i;
    private Dades f;    // és Serializable
}
```

Les classes a utilitzar per serialitzar i de-serialitzar objectes són **ObjectOutputStream** i **ObjectInputStream**, respectivament. Consulta la seva documentació, prestant especial atenció als mètodes d'escriptura i lectura.

La serialització presenta un problema de seguretat quan un membre d'una classe conté informació sensible a protecció de dades. En aquest cas, hi ha disponibles diverses tècniques per protegir aquesta informació. Fins i tot quan aquesta informació és privada (l'atribut siga *private*), una vegada que s'ha enviat al flux d'eixida algú podria llegir-la en l'arxiu en disc, o interceptar-la a la xarxa.

La manera més simple de protegir la informació sensible, com per exemple una contrasenya, és la de posar el modificador **transient** davant de l'atribut que la guarda. Per exemple, la següent classe *Client* té dos atributs: el nom del client i la contrasenya o password. Observa com aquest últim és marcat com *transient*. Anem a definir-hi la funció *toString()* tal que tornarà el nom del client i la contrasenya. En el cas que la contrasenya guardue el valor *null*, s'imprimirà el text "no disponible".

```
public class Client implements java.io.Serializable {
    private String nom;
    private transient String password;
    public Client (String nom, String pw) {
        this.nombre = nom;
        password = pw;
    }
    public String toString () {
        String text = (password == null)? "(No disponible)": password;
        text + = nom;
        return text;
    }
}
```

Vegem ara els passos per guardar un objecte de la classe *Client* a l'arxiu *clients.obj*. Posteriorment, es llegirà l'arxiu per reconstruir l'objecte *obj1* d'aquesta classe. Els passos per a la serialització són:

- Es crea l'objecte *Client* passant-li, per exemple, el nom del client "Miguel" i la contrasenya "xyz".
- Es crea un flux d'eixida (objecte *ObjectOutputStream*) i s'associa amb un objecte de la classe *FileOutputStream* per guardar la informació a l'arxiu *clients.obj*.
- S'escriu l'objecte client en el flux d'eixida mitjançant *writeObject*.
- Es tanca el flux d'eixida amb *close*.

El codi que realitza aquestes accions seria:

```
Client client = new Client("Angel","xyz");
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("clients.obj"));
oos.writeObject(client);
oos.close();
```

Per a reconstruir l'objecte de la classe *Client* es procedeix de la següent manera (deserialització):

- Es crea un flux d'entrada (objecte entrada de la classe *ObjectInputStream*) i s'associa amb un objecte de la classe *FileInputStream* per a llegir la informació que guarda l'arxiu *clients.obj*.
- Es llegeix l'objecte client en el flux d'eixida mitjançant *readObject*.

- S'imprimeix a pantalla aquest objecte cridant implícitament a *toString*.
- Es tanca el flux d'entrada cridant a *close*.

El codi corresponent és:

```
ObjectInputStream entrada = new ObjectInputStream(new FileInputStream("clients.obj"));
Client obj1 = (Client) entrada.readObject();
System.out.println("------------------------------");
System.out.println(obj1);
System.out.println( "------------------------------");
entrada.close();
```

L'eixida del programa és:

(No disponible) Angel

Aquesta eixida ens indica que la informació sensible, guardada a l'atribut *password*, no ha estat guardada a l'arxiu al ser marcada com *transient*. En la reconstrucció de l'objecte, amb la informació guardada a l'arxiu *password*, pren el valor *null*.

Punts a recordar per a la serialització:

1. Si una classe base implementa l'interfície *Serializable* les classes derivades no necessiten fer-ho, però no al contrari. Recorda que un objecte de la classe derivada també és objecte de la classe base; al contrari no sempre.
2. Només els membres no estàtics son guardats en la serialització.
3. Tampoc són guardats els membres marcats com a "transient".
4. Cap constructor d'objecte és cridat quan un objecte és deserialitzat.
5. Objectes inclossos com a atributs en la classe han d'implementar també l'interfície *Serializable*.

7 - CLASSES RELATIVES A FLUXOS

Interfície AutoCloseable

La instrucció *try-with-resources* (prova amb recursos) és una instrucció *try* que declara un o més recursos. Però què és un recurs?

Un recurs és un objecte que s'ha de tancar després que el programa haja acabat amb ell. La declaració de prova amb recursos assegura que cada recurs es tanca automàticament al final de la declaració. Qualsevol objecte que implemente *AutoCloseable* es pot utilitzar com a recurs.

El mètode close() d'un objecte AutoCloseable es crida automàticament en eixir del bloc *try*. L'ús de l'interfície garanteix un llançament ràpid, evitant excepcions d'esgotament de recursos i errors que es podrien produir d'una altra manera. Vegem un exemple.

```
public class BufferedReaderExemple
{
 public static void main(String[] args)
 {
  try ( FileReader fr = new FileReader("mostra.txt");
    BufferedReader br = new BufferedReader(fr); )
    {
       String sCurrentLine;

       while ((sCurrentLine = br.readLine()) != null)
            System.out.println(sCurrentLine);
```

```
    } catch (IOException e) {
    e.printStackTrace();
    }
 }
}
```

Altres classes de java.io.

Apart de les classes vistes fins ara, el paquet *java.io* conté un bon nombre d'altres classes que permeten el treball amb fluxos des d'altres estructures de dades, com poden ser canonades (pipes), arrays, etc.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

Per exemple, el següent programa extreu cada caràcter dels que componen un objecte String per mostrar-los en pantalla:

```
String s = "Això és un String";
StringReader sr = new StringReader(s);
int c = sr.read();
while (c! = - 1)
{
    System.out.println((char) c);
    c = sr.read();
}
```

També els podria escriure a un array:

```
String s = "Això és un String";
StringReader sr = new StringReader(s);
CharArrayWriter caw = new CharArrayWriter();
int c = sr.read();
while (c! = - 1)
```

```
{
    caw.write(c);
    c = sr.read();
}
```

A més de classes que permeten llegir o escriure en diferents estructures de dades, també tenim altres que permeten modificar el flux de dades. Per exemple, el següent codi transforma el text inicial substituint cada operador increment ++ per l'operació equivalent de sumar una unitat i assignar el valor resultant:

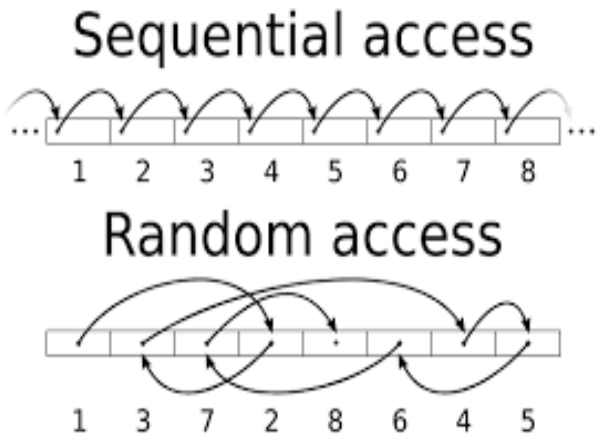
```
String s = "a++,c++;b+=5;c=a+b;b++;";
StringReader sr = new StringReader (s);
PushbackReader pbr = new PushbackReader(sr);
int ultim = pbr.read(), penultim = 0;
while (ultim! = -1)
{
    if (ultim == '+')
    // llegeix el següent i comprova si és un altre '+'
        if ((ultim = pbr.read()) == '+')
            System.out.print( "=" + (char) penultim + "+ 1");
        else
        {
            // si no era un altre '+', el retorne
            pbr.unread (ultim);
            System.out.print( '+');
        }
    else
        System.out.print ((char) ultim);
    penultim = ultim;
    ultim = pbr.read();
}
```

Aquest codi produirà l'eixida: *a=a+1,c=c+1;b+=5;c=a+b;b=b+1;*

8 - FITXERS D'ACCÉS DIRECTE

Per a treballar amb fitxers d'accés directe, o aleatori, Java inclou la classe **RandomAccessFile**. Accedeix a la documentació d'aquesta classe i, especialment, a la referida al mètode *seek*. Extret de la descripció de la classe:

"Les instàncies d'aquesta classe suporten tant la lectura com l'escriptura a un fitxer d'accés aleatori. Un fitxer d'aquest tipus es comporta com un gran array de bytes emmagatzemat en el sistema de fitxers. En ell tindrem alguna cosa similar a un cursor, o índex, al qual podem anomenar punter o marca de posició; les operacions d'entrada llegiran bytes començant des d'aquesta marca de posició i faran avançar aquesta fins al següent byte llegit. Si el fitxer aleatori és obert per a lectura i escriptura, les operacions d'escriptura es realitzaran a partir de la marca de posició i faran avançar aquesta tants bytes com el nombre de bytes escrits. Les operacions d'escriptura que actuen més enllà de la fi del fitxer implicaran la corresponent ampliació de l'arxiu. La situació de la marca de posició pot ser llegida pel mètode *getFilePointer* i canviada mitjançant el mètode *seek*".



Com a exemple inicial de l'ús d'aquesta classe es propossa el següent programa: codi que demane parells de valors numèrics (el primer sencer (posició) i el segon real (valor a escriure)) i escriga el valor en la posició indicada. L'usuari escriurà tants parells com vullga, fins a acabar amb una posició igual a zero.

Un ús més avançat d'aquesta classe, i d'aquest tipus de fitxers, podria ser un exemple de programa que gestione un fitxer de pel·lícules. Bàsicament farem que el programa treballe amb **formats de registre fixos**. Es a dir, per a facilitar el posicionament en cada pel·lícula farem que totes i cadascuna d'elles ocupen sempre la mateixa quantitat de bytes.

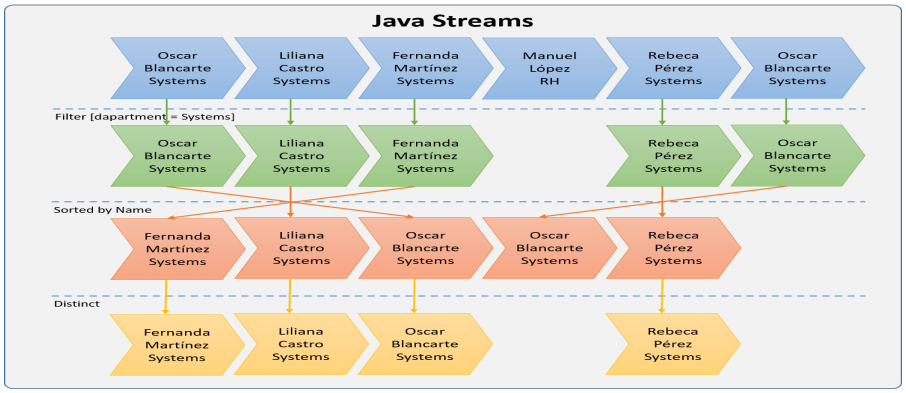
Cada pel·lícula, objecte *Film*, contindrà 4 camps: codi de pel·lícula, títol, any i director. Dos d'aquests camps són numèrics (enters, 4 bytes cada un, 8 en total) i els dos restants són objectes String. Atès que la mida d'aquests resultaria en principi variable, s'opta per donar-li una grandària fixa de 100 bytes a cadascun d'ells (en la majoria de casos, es desaprofiten bastants bytes, per exemple, per a "Titanic" sobrarien 93 bytes al ser de longitud 7). D'aquesta manera, totes les pel·lícules ocuparan exactament 210 bytes (101 cada String, és a dir, 100 caracters més un canvi de línia, i 4 cada enter: en total 101 + 101 + 4 + 4) i el contingut de l'arxiu haurà de ser llegit o escrit, des del principi de fitxer, avançant en múltiples d'aquesta quantitat. Per exemple, per escriure la pel·lícula amb codi 3, ens situarem en (codi - 1) * 210 bytes des del principi de l'arxiu: en aquest cas 2 * 210 = 420 bytes. Aquest serà el desplaçament que apliquem com a paràmetre al mètode de posicionament *seek*:

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

class Film
{
    private static final int maxString = 100;
    // 210 Bytes és el resultat de la següent operació
    private static final int tamFilm = 2*(maxString + 1) + 2*Integer.SIZE/8;
    private int cod;
    private String title;
    private int year;
    private String director;

    public Film(int c, String t, int i, String d)
    {
        cod = c; year = i;
        if (t.length() > maxString)
            title = t.substring(0, maxString);
        else
            title = t;
        if (d.length() > maxString)
            director = d.substring(0, maxString);
        else
```

```
            director = d;
        }

    public void show()
    {
        if (cod ! = 0)
            /* La funció printf usa els especificadors de format
                    %d per a enter,
                    %s per a text (string) segons el tipus de la variable a la que repre
                    4 especificadors de format per a 4 variables, entre canvis de línia
            System.out.printf("\n%d\t%s\n\t%d\tDirector:%s\n---\n", cod, title, year, di
        else
            System.out.println("Aquesta pel·lícula no existeix");
    }

    public void writeFile(RandomAccessFile raf)
    {
        try
        {
            // situar la marca de posició on corresponga segons el codi
            raf.seek((cod - 1)*tamFilm);
            // ara escric les dades de la pel·lícula a fitxer
            raf.writeInt(cod);
            raf.writeBytes(title + '\n');
            raf.writeInt(year);
            raf.writeBytes(director + '\n');
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    public void readFile (RandomAccessFile raf)
    {
        try
        {
            // situar la marca de posició on corresponga segons el codi
            raf.seek((cod-1)*tamFilm);
             // ara llig les dades de la pel·lícula
            int cod = raf.readInt();
            if (cod ! = 0)  // comprove que realment hi han dades
            {
                title = raf.readLine();
                year = raf.readInt();
                director = raf.readLine();
            }
        }
        catch (EOFException i)
        {
            System.out.printf("Aquesta pel·lícula no existeix\n");
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
```

```
    }

public int getYear() { return year; }
}

    public class films
    {
        public static void main (String [] args) throws FileNotFoundException, IOExcepti
        {
            Scanner ent = new Scanner (System.in);
            RandomAccessFile raf = new RandomAccessFile("films.dat", "rw");
            System.out.printf ( "GESTIÓ VIDEOTECA\n\t1. Introduir pel·lícula\n
                \t2. Consultar pel·lícula\n\t0. Eixir \nTria opció: \n");
            int opc = ent.nextInt ();
            int c; Film f;
            switch (opc)
            {
                case 1: System.out.printf("Introdueix codi:");
                c = ent.nextInt ();
                // problema de la memòria intermèdia del teclat
                ent.nextLine();
                System.out.printf("Introdueix títol:");
                String t = ent.nextLine();
                System.out.printf("Introdueix any:");
                int i = ent.nextInt();
                // problema de la memòria intermèdia del teclat
                ent.nextLine();
                System.out.printf("Introdueix director:");
                String d = ent.nextLine();
                f = new Film(c, t, i, d);
                f.writeFile(raf);
                break;

                case 2: System.out.printf( "Introdueix codi:");
                c = ent.nextInt();
                f = new Film(c, "", 0, "");
                f.readFile(raf);
                if (f.getYear() != 0)
                    f.show();
                break;

                default: if (raf != null)
                                raf.close();
                            System.out.printf( "Fi del programa");
            };

}
    }
```

9 - INTERFICIE STREAM



La versió Java 8 va introduir canvis revolucionaris al llenguatge, sent els *Streams* i la programació funcional els dos més significatius. Els elements de programació funcional afegits al llenguatge són:

- Interfícies funcionals: interfícies amb un sol mètode abstracte (com *Predicate*, *Function*, *Consumer*).
- Expressions lambda: sintaxi concisa per implementar interfícies funcionals.

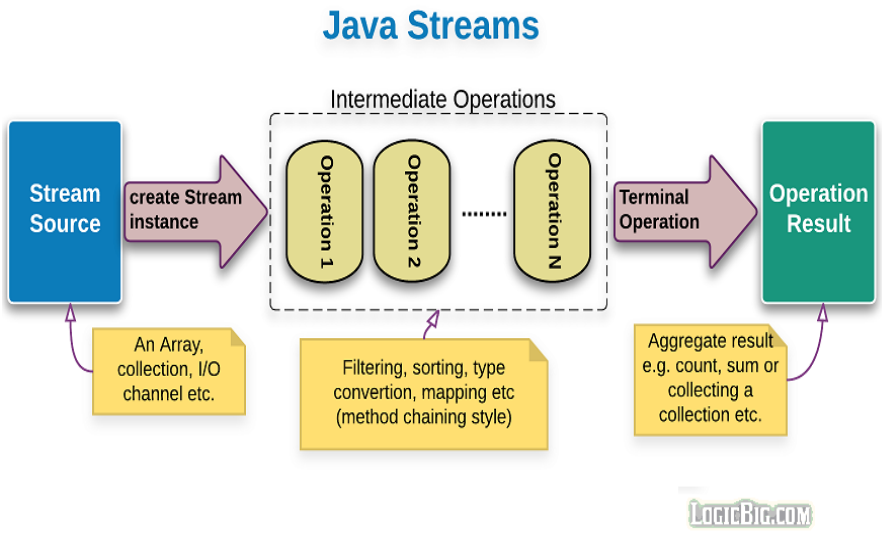
Els *Streams* són un conjunt de mètodes i classes per a processar col·leccions (semblant als arrays estàtics o als objectes *Vector* o *ArrayList*) però de manera funcional. En general milloren la llegibilitat i l'eficiència de moltes operacions amb conjunts de dades. La conversió d'arrays, o altres coleccions de dades, en fluxos també augmenta el rendiment general del programa. A més d'això, també podem utilitzar els diversos mètodes de l'*API Stream* que poden simplificar el mapatge (conversió de les dades) i les accions de filtratge en arrays (esborrar elements). L'*API Stream* de Java permet convertir col·leccions com els arrays, estàtics o dinàmics, en fluxos. Inclús poden processar cada element en paral·lel.

Extret de la documentació per a aquesta interfície: Un *Stream* és una seqüència d'elements que suporten operacions agregades, seqüencials i paral·leles. El següent exemple il·lustra una operació agregada utilitzant *Stream* i *IntStream*:

```
Vector<Figura> figures = new Vector<Figura>();
// carreguem objectes Figura en el vector figures
// ...
double suma = figures.stream()
        .filter(w -> w.getColor() == RED)       // operació intermèdia
        .mapToInt(w -> w.getWeight())           // operació intermèdia
        .sum();                                 // operació final o terminal
```

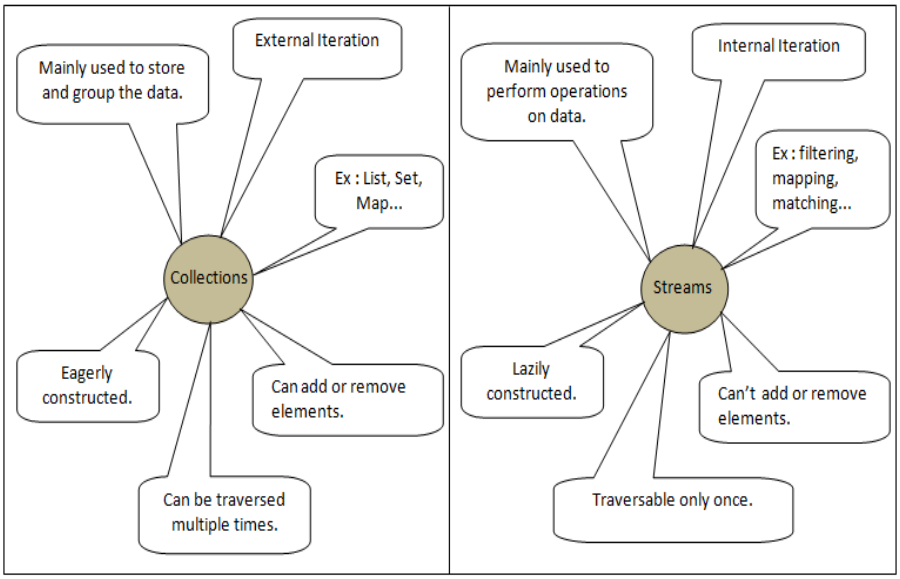
En aquest exemple, figures és una *Collection<Figura>*, com podria ser un objecte *ArrayList* o *Vector*. Creem un flux d'objectes *Figura* amb *Collection.stream()*, ho filtrem per a produir un flux que continga només els components rojos (RED) i després el transformem en un flux de valors enters que representen el pes de cada figura roja. Després, aquest flux se suma per a produir un pes total.

A més de *Stream*, existeixen especialitzacions primitives per a *IntStream*, *LongStream* i *DoubleStream*, totes les quals es denominen "fluxos" i s'ajusten a les característiques i restriccions descrites ací.



Per a realitzar un càlcul, les operacions de flux es componen en una canalització de flux. Aquesta canalització consta d'una font o **origen** (que pot ser un array, una col·lecció, una funció de generador, un canal d'E/S, etc.), zero o més **operacions intermèdies** (que transformen un flux en un altre flux, com a filtre (usant un predicat)) i una **operació terminal** (que produeix un resultat o un efecte secundari, com *count()* o *forEach(Consumer)*).

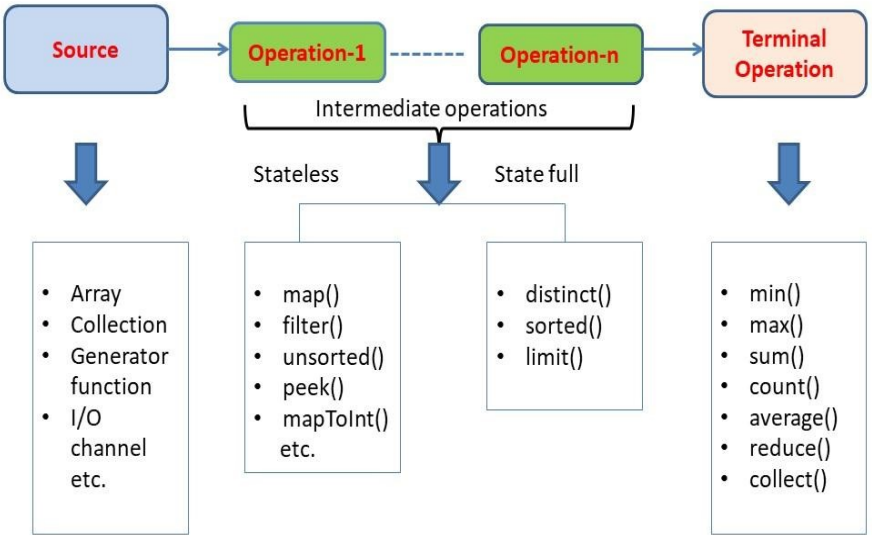
Les col·leccions i els fluxos, si bé tenen algunes similituds superficials, tenen objectius diferents. Les col·leccions s'ocupen principalment de la gestió eficient i l'accés als seus elements. Per contra, els fluxos no proporcionen un mitjà per a accedir o manipular directament els seus elements i, en canvi, es preocupen per descriure declarativament la seua font i les operacions computacionals que es realitzaran en conjunt en aqueixa font. No obstant això, si les operacions de flux proporcionades no ofereixen la funcionalitat desitjada, les operacions *BaseStream.iterator()* i *BaseStream.spliterator()* es poden usar per a realitzar un recorregut tradicional per els seus elements.



La majoria de les operacions en un flux accepten paràmetres que descriuen el comportament especificat per l'usuari, com l'expressió lambda *w -> w.getWeight()* passada a *mapToInt* en l'exemple anterior. Per a preservar el comportament correcte, aquests paràmetres de comportament han de complir:

- no han d'interferir (no modifiquen la font o origen del flux)
- en la majoria dels casos, ha de ser sense estat (el seu resultat no ha de dependre de cap estat que puga canviar durant l'execució del flux).

Aquests paràmetres són sempre instàncies d'una interfície funcional com *Function* i, generalment, són expressions lambdes o referències a mètodes. Llevat que s'especifique el contrari, aquests paràmetres no han de ser nuls.



Els fluxos tenen un mètode *BaseStream.close()* i implementen *AutoCloseable*, però quasi tots els fluxos no necessiten tancar-se després del seu ús. En general, només serà necessari tancar els fluxos l'origen dels quals siga un canal IO (com les que retorna *Files.lines(Path, Charset)*).

Les canalitzacions de flux poden executar-se seqüencialment o en paral·lel. Aquesta manera d'execució és una propietat de la seqüència. Els fluxos es creen amb una opció inicial d'execució seqüencial o paral·lela. (Per exemple, *Collection.stream()* crea un flux seqüencial i *Collection.parallelStream()* crea un paral·lel). Aquesta elecció de manera d'execució pot ser modificada pels mètodes *BaseStream.sequencial()* o *BaseStream.parallel()*, i es pot consultar amb el mètode *BaseStream.isParallel()*.

```java
import java.util.stream.Stream;

public class MainJava {

    public static void main(String[] args) {
        Stream.of(1, 3, 6, 2, 4)
                .takeWhile(i -> i < 4)
                .forEach(System.out::println);
    }
}
```

El següents mètodes estàtics de la classe *Arrays* es poden utilitzar per convertir un array, passat com a paràmetre, en un flux.

Stream stream(T[ ] array)

IntStream stream(int[ ] array)

LongStream stream(long[ ] array)

DoubleStream stream(double[ ] array)

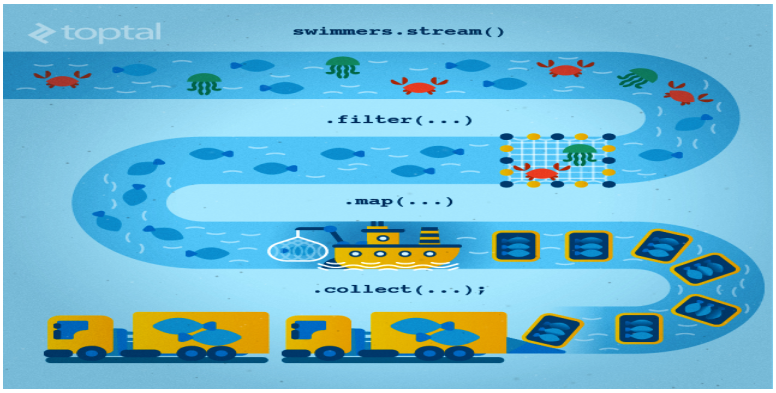
El programa següent mostra la implementació de l'ús de fluxos amb arrays. Aquest programa ens permet mostrar una comparació de l'enfocament iteratiu i l'enfocament de fluxos.

```
    // Programa que mostra la suma de les xifres unitats dels valors d'un array
import java.util.Arrays;
class Main {
    public static void main(String[] args)
        {
            // sumarem 5 + 4 + 7 + 0 + 4 + 1 + 3 + 1 (les unitats de cada valor)
            int intArray[] = {15,324,17,20,124,301,33,141};
            // calcular la suma usant iteració tradicional
            int sum = 0;
            for (int i = 0; i < intArray.length ; i++)
                sum += intArray[i]%10;
            System.out.print("Suma: " + sum));
        }
    }
```

El problema d'aquestes solucions és que a mesura que creix l'array, l'enfocament iteratiu es fa més lent. El següent exemple de programació mostra la mateixa funcionalitat feta amb *streams*.

```
import java.util.Arrays;
class Main {
    public static void main(String[] args)
     {
         //declare array of ints
         int[] ints = {15,324,17,20,124,301,33,141};
        // mapeja el flux d'enters a les unitats i calcula la suma
        int sum = Arrays.stream(ints).map(i -> i % 10).sum();
        // mostra la suma
        System.out.println("La suma de les unitats és: " + sum);
     }
}
```

El mètode *map()* aplica una funció a cada element d'un flux o el transforma a a un valor diferent. Es tracta d'un mètode intermediari, mentre que *sum()* és un mètode terminal.



¿La solució amb fluxos és més ràpida que la del bucle Java? Sí, sobretot amb fluxos paral·lels. Per exemple, el mètode *parallelSort()* de la classe *Arrays*, que utilitza fluxos paral·lels, és més ràpid que el mètode d'ordenació seqüencial proporcionat per la classe *Arrays*.

10 - PAQUET JAVA.NIO

En la versió 7 de Java es va introduir el paquet *java.nio.file*, donant pas a l'anomenada NIO2, la qual va suposar una important millora respecte a l'antic maneig de fitxers. Operacions aparentment tan senzilles com llegir/escriure un arxiu de text en/des d'un String suposaven l'escriptura de diverses línies de codi. Previament, en la versió 4 del llenguatge s'introdueix la nova API IO, inclosa al paquet *java.nio* (nio significa nou IO). Aquest paquet va introduir tres classes bàsiques: *Channel*, *Buffer* i *Selector*.

En la web d'Oracle tenim una equivalència de funcionalitats entre la classe *File* de *java.io* i el paquet *java.nio.file*: https://docs.oracle.com/javase/tutorial/essential/io/legacy.html#mapping.

Es mostraran a continuació diversos exemples típics d'ús, on es veurà la potència i senzillesa d'aquest paquet. Comencem per un exemple que mostra tot el contingut d'un fitxer fent ús de la interfície *Stream*.

Observa la utilitat dels mètodes estàtics de les classes *Paths* i *Files*. Observa com treballem amb el fitxer amb una instància de la classe *Path* (a distinguir de la classe *Paths*) per a mostrar totes les línies d'un fixer.

```
import java.io.*;
import java.nio.*;
import java.nio.file.*;
import java.util.stream.*;

public class ex1 {
        public static void main(String args[]) throws IOException
        {
                Path input = Paths.get("text.txt");
                Stream<String> st = Files.lines(input);
                // st.forEach( s -> System.out.println(s) );
                st.forEach(System.out::println);    // equivalent a l'anterior línia
        }
}
```

Crea un fitxer buid si encara no existeix. Observa l'ús d'altres mètodes estàtics de **Files**.

```
Path emptyFile = Paths.get("exemples/myFile.txt");
if (Files.notExists(emptyFile))
    emptyFile = Files.createFile(emptyFile);
```

Llig el contingut d'un fitxer de text en una cadena (tot d'una lectura amb el mètode estàtic *readAllBytes*.

```
String content = new String(Files.readAllBytes(Paths.get("exemples/sampleText.txt")),
        StandardCharsets.UTF_8);
System.out.println(content);
```

Llig el contingut d'un fitxer de text línia a línia (amb *readAllLines*).

```
List<String> lines = Files.readAllLines(Paths.get("exemples/sampleText.txt"),
        StandardCharsets.UTF_8);
for (String line : lines)
    System.out.println(line);
```

Escriu un String a un fitxer de text, sobreescrivint-ho si existira amb les corresponents opcions d'*StandardOpenOption*).

```
String text = "Això és una cadena de prova\n";
Files.write(Paths.get("exemples/writeText.txt"), text.getBytes(StandardCharsets.UTF_8),
        StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
```

Escriu una llista de String a un fitxer de text, sobreescrivint-ho si ja existira

```
List<String> textLines = Arrays.asList("Línia 1", "Línia 2", "Línia 3");
Files.write(Paths.get("exemples/writeLines.txt"), textLines, StandardCharsets.UTF_8,
        StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
```

Crea una estructura de directoris de forma recursiva. Si algún directori ja existira no llançarà excepció

```
Files.createDirectories(Paths.get("exemples/level1/level2/level3"));
```

Mida en bytes d'un directori. En aquest cas utilitza el mètode *walk* que genera un fluxe (stream) de fitxers del directori "exemples", el transfoma en un fluxe paralel, filtra cadascun d'ells que siguen fitxers regulars amb *filter* i mapeja o transforma el fitxer en la seua longitud, per a acabar sumant totes les longituds.

```
long size = Files.walk(Paths.get("exemples")).parallel()
        .filter(p -> p.toFile().isFile())
        .mapToLong(p -> p.toFile().length()).sum();
System.out.println("Mida del directori: " + size);
```

Compta el nombre de fitxers d'un directori (excloent subdirectoris).

```
long count = Files.walk(Paths.get("exemples")).parallel()
        .filter(p -> !p.toFile().isDirectory()).count();
System.out.println("Total de fitxers: " + count);
```

Llista recursiva amb els directoris continguts en un directori pare

```
List<Path> dirs = Files.walk(Paths.get("exemples")).filter(Files::isDirectory)
        .map(x -> x.toAbsolutePath()).collect(Collectors.toList());
for (Path dir : dirs)
    System.out.println("Ruta del directori: ".concat(dir.toString()));
```

Llista recursiva amb els fitxers continguts en un directori

```
List<Path> files = Files.walk(Paths.get("exemples"))
        .filter(Files::isRegularFile).map(x -> x.toAbsolutePath())
        .collect(Collectors.toList());
for (Path file : files)
    System.out.println("Ruta del fitxer: ".concat(file.toString()));
```

Neteja el contingut d'un directori esborrant de forma recursiva tots els fitxers continguts al mateix

```
Files.walk(Paths.get("to_be_cleared")).parallel().filter(p -> p.toFile().isFile())
        .forEach(File::delete);
```

Esborra de forma recursiva un directori i tot el seu contingut

```
Files.walk(Paths.get("to_be_deleted")).sorted(Comparator.reverseOrder())
        .map(Path::toFile).forEach(File::delete);
```

Còpia un directori amb tot el seu contingut des d'un origen a un destí

```
Path from = Paths.get("exemples/source_dir");
Path dest = Paths.get("exemples/dest_dir");
try (Stream<Path> stream = Files.walk(from)) {
    stream.forEachOrdered(source -> {
        try {
            Files.copy(source, dest.resolve(from.relativize(source)),
                 StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
}
```

Mou un directori amb tot el seu contingut

```
Files.move(Paths.get("exemples/source_dir"), Paths.get("examples/dest_dir"),
        StandardCopyOption.REPLACE_EXISTING);
```

Converteix de *Path* a *File* i de *File* a *Path*

```
File file = onePath.toFile();
Path path = oneFile.toPath();
```

Com s'ha pogut comprovar, l'ús d'aquesta nova API permet navegar, llegir i escriure fitxers, consultar els seus atributs (dates de creació, modificació, accés, permisos…) copiar i moure fitxers o directoris, reconéixer enllaços simbòlics, entrada/eixida asíncrona i més funcionalitats que ens facilitaran enormement el treball amb fitxers i directoris sense necessitat d'importar llibreries externes.