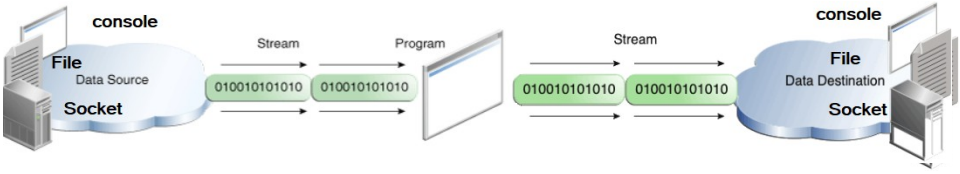
# UNIDAD 7

## STREAMS Y FICHEROS




BIBLIOGRAFÍA:

Java, Cómo Programar (10 Ed) Pearson.Paul Deitel&Harvey Deitel (2016).
Introduction to Programming Using Java (7ª Ed.). David J. Eck (2014)
Java The Complete Reference (7 Ed). Hervert Schildt. McGraw (2007)

# 7.1. STREAMS (FLUJOS).

Sin la habilidad para interactuar con el resto del mundo, un programa sería menos útil. Esta habilidad consiste en la capacidad para intercambiar información (entrada y salida, IO en inglés).

Un programa necesitará intercambiar información con muchos dispositivos de entrada y salida diferentes. Si el programa debe ocuparse de los detalles de cada uno de ellos, la tarea se vuelve agotadora. Por eso se han creado abstracciones para representar estos dispositivos y ocultar los detalles de cada uno. En Java, la principal abstracción se llama **stream** y representa el movimiento de información de un lugar a otro (de un origen a un destino). También hay otras como **ficheros** y **canales**. Un movimiento de información que entra al programa es un stream de entrada o si sale de él, de salida.

La mayor parte de las clases de streams estarán en el paquete `java.io`. Los ficheros pueden ser el origen o el destino de la información que mueve un stream (aunque no es el único origen o destino).

Las operaciones con ficheros pueden generar errores en tiempo de ejecución del tipo `IOException` que tendremos que tratar en nuestros programas.

## 7.1.1. STREAMS DE CARACTERES Y DE BYTES.

Hay dos formas básicas de representar los datos: entendibles por los humanos (caracteres) o como los manejan las máquinas (bytes). Si el número PI lo envías de forma entendible para un humano, será una serie de caracteres que interpretas como un valor numérico (3.141592654) pero si lo mandas como lo representa una máquina, serán una serie de bits, que formarán bytes, pero que un humano no entiende si lo visualiza directamente. Java tiene los dos tipos de

formatos, streams de caracteres y streams de bytes.

Las clases streams se definen en el paquete **java.io**, debes importar las clases individuales o todas para usarlas:

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

Lo bueno de los streams es que resulta fácil intercambiar datos (leer/escribir) con un fichero, una red o la pantalla, es igual. Estas clases abstractas (**Reader**, **Writer**, **InputStream**, **OutputStream**) y sus subclases, ofrecen operaciones de E/S muy básicas (de bajo nivel).

Por ejemplo la clase **InputStream** declara un método de instancia abstracto (**public int read() throws IOException**) para leer un byte (entero en el rango 0 a 255) de un stream de entrada. Si se encuentra el final del stream, devuelve el valor -1. Si ocurre algún error, se lanza la excepción de tipo **IOException**.

Esto significa, que siempre deberías usar la operación dentro de un bloque **try-catch-finally** o equivalente o bien en un método que declare una cláusula **throws IOException**.

## STREAMS DE CARACTERES

Las clases abstractas **Reader** y **Writer** mueven caracteres. Si un programa escribe un número usando una sublase suya, hay que traducirlo a caracteres antes, igual ocurre cuando los lee (los programas Java trabajan con caracteres representados como valores Unicode de 16-bits y las personas que usan alfabetos occidentales, lo hacen en código ASCII de 8 bits cada carácter). Las clases streams se ocupan de esta traducción y del código de caracteres a emplear de manera automática.

Un programa que use streams de caracteres en vez de streams de

bytes adapta automáticamente al conjunto de caracteres local los valores sin ningún esfuerzo extra para el programador. Si la internacionalización de tu aplicación no es una prioridad, no debes preocuparte por nada más, en otro caso no te costará excesivo esfuerzo realizarla.

Otra característica interesante de los streams, es que unos pueden envolver a otros, es decir, si el trabajo que ofrece un stream básico (mover caracteres y trasladar códigos) no es suficiente para las necesidades de tu programa, puedes envolverlo en otro de más alto nivel (que interprete los caracteres como líneas y palabras por ejemplo), de esta forma, lo que el stream envuelto envía o recibe se lo pasa al que hace de envoltura, para que le aplique sus habilidades.

En la imagen vemos una cadena de algunos hijos de las clases abstractas **`Printer`** (`PrintWiter`, `FileWriter`) y `Reader` (`Scanner`, `FileReader`) formando una especie de cadena de streams. La primera fila forma un streams de salida (mueve caracteres desde el programa a un fichero). El programa envía Strings a un fichero, el stream `PrintWriter` envuelve a un `FileWriter` (es más básico, de más bajo nivel, mueve caracteres).

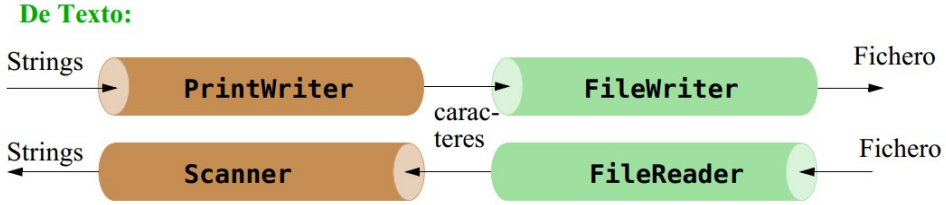



***Figura 1: streams de texto encadenados.***

**EJEMPLO 1**: Un programa que escribe cadenas a un fichero de caracteres llamado "fichero.txt" usando una envoltura (el programa envía cadenas a un `PrintWriter` que envuelve a un `FileWriter`, que envía

caracteres a un fichero).

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class E1{
   public static void main(String[] args) throws IOException {
       FileWriter fw = new FileWriter("fichero.txt");
       PrintWriter pw = new PrintWriter( fw );
       String nombre = "María Luisa";
       int edad = 21;
       double sueldo = 1234.56;
       pw.println("Nuestro candidato:");
       pw.format("  {nombre: \"%s\", edad: %d, sueldo: %.2f}",
                 nombre, edad, sueldo);
       pw.close();
   }
}
```

Los streams, una vez que se han utilizado y ya no son necesarios hay que cerrarlos. Si cierras el que envuelve a otros, se cerrarán todos. Como consumen recursos, cuanto antes se haga mejor. Y como dejarlos abiertos (tras un error) supone perder recursos, se usan sentencias try-catch con recursos que automatizan el finally. Repasa la unidad 3.

Al principio Java no tenía streams de caracteres, codificaba caracteres solamente en ASCII y los streams de bytes hacían casi el mismo papel. Por ejemplo: **System.in** y **System.out** son streams de bytes.

Las clases **Reader** y **Writer** (sus clases hijas en realidad) aportan métodos muy básicos de lectura y escritura de caracteres en vez de bytes ( si c es un carácter: **write(c)** y **char c = read()** ).

**EJERCICIO 1**: Haz un programa que pregunte por el nombre de un fichero y lea repetidamente cadenas de texto hasta que se teclee una cadena vacía. Cada vez que lea una cadena, que la envíe al fichero

mediante un stream de caracteres de tipo PrintWriter.

Las clases orientadas al flujo de bytes nos proporcionan la suficiente funcionalidad para realizar cualquier tipo de operación de entrada o salida, pero no pueden trabajar directamente de manera cómoda para el programador con caracteres Unicode, los cuales están representados por 2 bytes. Por eso, se consideró necesaria la creación de las clases orientadas al flujo de caracteres.

Como ya se ha comentado Java dispone de 2 clases abstractas (**Reader y Writer**) y todas sus subclases indican en el constructor el objeto que representa el origen o el destino del flujo de datos.

Hay que recordar que cada vez que se llama a un constructor se abre el flujo de datos y es necesario cerrarlo cuando no lo necesitemos.

Existen muchos tipos de flujos dependiendo de la utilidad que le vayamos a dar a los datos que extraemos de los dispositivos. Un flujo básico como un FileWriter puede ser envuelto por otro flujo para hacer más sencilla o más eficiente su uso. Así, un **BufferedWriter** nos permite manipular el flujo de datos con un buffer (ganamos eficiencia), pero si lo envolvemos en un **PrintWriter** lo podemos escribir con muchas más funcionalidades adicionales para diferentes tipos de datos (subimos de nivel) y nos hace más sencillo cierto tipo de trabajos.

```
PrintWriter pw = new PrintWriter(
                        new BufferedWriter(
                                new FileWriter("fichero.txt")
                            )
                    );
```

En el trozo de código de ejemplo se ve cómo podemos escribir la salida estándar a un fichero.


Si observas el ejemplo 2, verás cómo se usa **InputStreamReader** que es un stream puente entre bytes y caracteres: lee bytes y los decodifica a caracteres. **BufferedReader** lee texto de un flujo de entrada de caracteres, permitiendo efectuar una lectura eficiente de caracteres, arrays y líneas, es decir, sube el nivel (distingue grupos de caracteres) y sabe manejar buffers. Y usa **FileWriter** para flujos de caracteres, pues para datos binarios se utiliza **FileOutputStream**.

**EJEMPLO 2**: Un trozo de código que escribe cadenas de texto a un fichero de caracteres "\prueba\salida.txt" usando varias envolturas y un try-catch con recursos para segurar el cierre automático de los streams.

```
try(
   PrintWriter pw = new PrintWriter(
                        new FileWriter("\\prueba\\salida.txt",true));
   BufferedReader br = new BufferedReader(
                           new InputStreamReader(System.in));
){
   String s;
   while ( !(s = br.readLine()).equals("salir") ){
      pw.println(s);
   }
   out.close();
}
catch(IOException ex){
   System.out.println( ex.getMessage() );
}
```

## RUTAS DE SISTEMAS DE FICHEROS

**Rutas absolutas y relativas**: cuando escribimos un fichero podemos indicar solo su nombre o además del nombre la ruta que ocupa en el

sistema de ficheros (**pathname**). Las rutas pueden indicarse de manera absoluta si comenzamos desde la carpeta raíz de un disco (por ejemplo "c:\\Usuarios\\manolo\\escritorio\\fichero.txt" en Windows o "/home/manolo/escritorio/fichero.txt" en GNU/Linux) o bien de manera relativa si comenzamos el camino a seguir para localizarlo desde la carpeta actual, indicando la secuencia de movimientos a seguir ("fotos\\viaje\\paisaje1.jpg" en Windows o "./fotos/viaje/paisaje2.jpg" en GNU/Linux).

**Separadores de carpetas:** Si usas el separador de carpetas que usa Windows y MS-DOS que es la contrabarra (\), debes escapar el significado del carácter '\' en una cadena que se utiliza como carácter de escape. Para escapar del escape hay que escribirlo dos veces para que no se interprete como el escape: "\\". Por ejemplo la siguiente ruta: "carpeta\numeros.txt" sería un pathname de fichero erróneo porque se interpreta como "carperta" + [salto_línea] + "umeros.txt". La manera correcta de escribir la ruta sería "carpeta\\numeros.txt".

Si usas el separador de GNU/Linux (/), aunque el programa se ejecute en Windows, Java automáticamente lo convertirá al delimitador adecuado. Es decir, la ruta: "C:/copias/peticion.docx" se interpretaría como "C:\\copias\\peticion.docx" automáticamente. Al contrario no pasa, es decir, si escribes una ruta de Windows, en Linux no funcionaría.

Aunque en muchos de los ejemplos de la unidad se use la ruta de los ficheros tal y como se usan en Windows, será preferible usar la nomenclatura de Linux (que funciona tanto en uno como en otro sistema). También se recomienda usar en Java la constante definida en **File.separator.**

**Separar ruta y nombre de un pathname:** a veces cuando se manipulan pathnames a ficheros surgen problemas debido a la gran cantidad de

posibilidades. Por ejemplo, si preguntamos por consola el nombre de un fichero con el que trabajar, podrían teclear solo el nombre del fichero o pueden teclear la ruta y el nombre. Para separarlos podemos usar los métodos de la clase **java.io.File** como son **getName()** y **getParent()** y otros.

```
String fichero1 = "carta.docx";  // fichero solo con nombre
String fichero2 = "./pendiente/febrero/carta.odt"; // fichero con ruta + nombre
// separar ruta y nombre de fichero1
File f1 = new File(fichero1);
String nombre1 = f1.getName();  // nombre1 contiene "carta.docx"
String ruta1 = f1.getParent();  // ruta1 vale null
// separar ruta y nombre de fichero2
File f2 = new File(fichero2);
String nombre2 = f2.getName();  // nombre2 contiene "carta.odt"
String ruta2 = f2.getParent();  // ruta2 contiene "./pendiente/febrero"
```

## STREAMS DE BYTES

Las clases abstractas **InputStream** y **OutputStream** mueven bytes. El intercambio de información en forma de bytes será rápido (no hacen conversiones) y en general los datos pesan menos, pero no son entendibles por los humanos y pueden aparecer incompatibilidades si las máquinas/sistemas de origen y destino son diferentes. En la figura 2, como los FileInputStream y FileOutputStream solo mueven bytes, se envuelven en ObjectInputStream y ObjectOutputStream para que el programa pueda mover tipos de datos o bien objetos directamente.

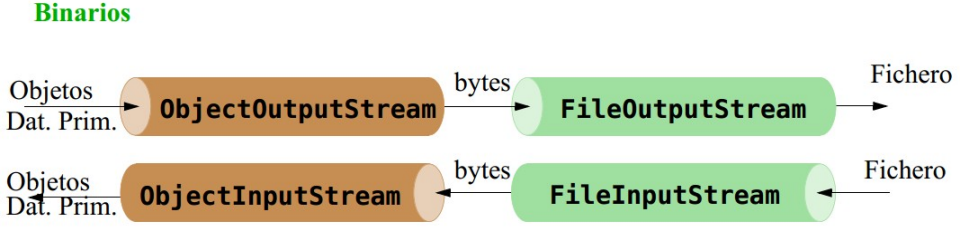



*Figura 2: streams binarios encadenados.*

La clase **InputStream** define un método para leer un byte (**int b =**

**is.read()**) y otro método para leer varios bytes de datos de una sola vez y almacenarlos en un array de bytes (**int bytesLeidos = is.read(byte[])**). Pero no ofrece otros métodos para leer tipos de datos como un **int** o un **double**.

Tampoco es un problema porque nunca crearás un objeto de tipo **InputStream**. Usarás subclases que añaden métodos más elaborados para leer datos. Con la clase **OutputStream** ocurre algo similar, define un método abstracto para escribir datos (**write(int b)**), donde el parámetro es int pero el valor sufre un casting a byte antes de moverlo (se hace así para evitar problemas de signo) y un método para volcar el contenido de un buffer: **write(int[] b, int desde, int longitud)**.

Este tipo de flujo es el idóneo para el manejo de entradas y salidas de bytes y su uso por tanto está orientado a la lectura y escritura de datos binarios. Para el tratamiento de los flujos de bytes, Java tiene dos **clases abstractas** (**InputStream** y **OutputStream**) y cada una de estas clases abstractas tiene varias subclases que controlan las diferencias entre los distintos dispositivos de E/S que se pueden utilizar.

```
class FileInputStream extends InputStream {
   FileInputStream(String fichero) throws FileNotFoundException;
   FileInputStream(File fichero) throws FileNotFoundException;
   ...
}

class FileOutputStream extends OutputStream {
   FileOutputStream(String fichero) throws FileNotFoundException;
   FileOutputStream(File fichero) throws FileNotFoundException;
   ... ... ...
}
```

**OutputStream** e **InputStream** y todas sus subclases, reciben en el constructor un parámetro que indica el objeto que representa el origen o el destino del flujo y que es el nombre de un fichero o un fichero.

**EJEMPLO 3**: Método que recibe dos nombres de ficheros e intenta copiar el primero en el segundo.

```
public void copiar(String origen, String destino) {
   FileInputStream entrada = null;
   FileOutputStream salida = null;
   try {
      entrada = new FileInputStream(origen);
      salida = new FileOutputStream(destino);
      byte[] buffer= new byte[256];  // Usar buffer para mejorar rendimiento
      while (true) {
         int n = entrada.read(buffer); // Leer el flujo de bytes
         if(n < 0)
            break;  // Si no queda nada por leer, salir
         salida.write(buffer, 0, n);  // Escribir n bytes en salida
      }
   } catch(IOException e) {
      System.out.println( e.getMessage() );
   } finally { // Cerrar los ficheros
      entrada.close();
      salida.close();
   }
}
```

## CLASES DE JAVA.IO

Las clases del paquete **java.io** se pueden ver en la figura 3. Destacamos las clases relativas a flujos:

- **FileInputStream**: lee bytes de un fichero. Es binario, de entrada y de bajo nivel. Tiene un constructor que recibe el nombre del fichero y otro que recibe un `File`. Para leer la información tiene un método que devuelve un byte: `int dato = r.read()`. Este método devuelve -1 si ya no quedan bytes por leer. Y tiene otro `int cantidad = read(byte[] buffer)` que intenta rellenar de bytes el array buffer y devuelve la cantidad de bytes que ha movido al buffer y -1 si no quedan más bytes por mover. Tiene otros métodos como `available()` que devuelve la cantidad de bytes que quedan por leer.

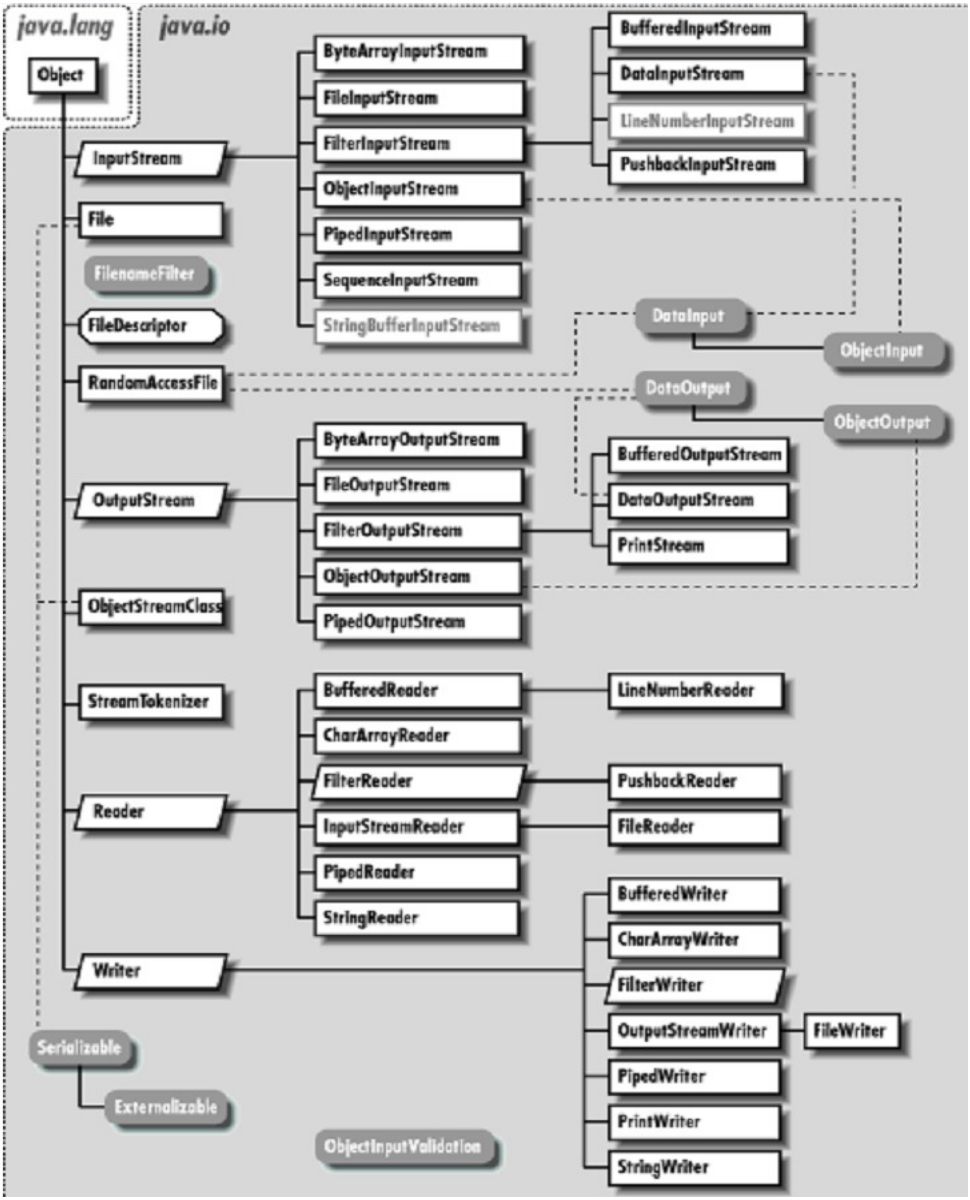



*Figura 3: Clases e interfaces de java.io.*

- **FileOutputStream:** escribe bytes en un fichero.
- **BufferedInputStream**: flujo que lee bytes y sabe usar buffers.
- **BufferedOutputStream**: escribe bytes y sabe usar buffers.
- **StringReader:** flujo de caracteres cuyo origen es un String.
- **StringWriter:** flujo de caracteres cuyo destino es un string.
- **BufferedReader:** lee cadacteres usando buffer.
- **BufferedWriter:** escribir caracteres usando buffer.
- **StreamTokenizer**: lee un flujo de caracteres de entrada, lo analiza (parse) y divide en trozos (tokens), permite manipualr tokens.

## ENVOLVER STREAMS DE BYTES CON STREAMS DE CARACTERES

A veces es conveniente envolver streams de bytes con streams de caracteres. El stream de caracteres realiza la traducción del conjunto de caracteres del fichero al conjunto del programa o viceversa y el stream de bytes se ocupa de la I/O física. Por ejemplo, FileReader puede usar un FileInputStream y FileWriter usar un FileOutputStream.
Hay otros dos streams puente de byte --> carácter que son **InputStreamReader** y **OutputStreamWriter**. Se usan por ejemplo en sockets, donde lo que transporta un paquete de red son bytes pero a tu programa igual le interesa trabajar con caracteres.

Aunque nos centramos en mover datos a ficheros, lo cierto es que las clases de streams de io como **charArrayReader**, **charArrayWriter**, **StringReader**, **StringWriter**, **byteArrayInputStream** y **byteArrayOutputStream** nos permiten manipular fácilmente buffers.

**EJEMPLO 4**: fragmento de progama de un cliente de red que lee y escribe en un socket.

```
String host = args[0];                    // IP o nombre DNS
```

```
int puerto = Integer.parseInt(args[1]);  // nº puerto
try (
    Socket ecoS = new Socket(host, puerto);
    PrintWriter out =  new PrintWriter(ecoS.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
                        new InputStreamReader(ecoS.getInputStream()));
    BufferedReader stdIn = new BufferedReader(
                               new InputStreamReader(System.in))
){
    String entrada;
    while ( (entrada = stdIn.readLine() ) != null) {
        out.println(entrada);
        System.out.println("eco: " + in.readLine() );
   }
} catch(IOException e){}
```

## ENVOLVER STREAMS CON BUFFERED STREAMS

Con **BufferedInputStream** podemos añadir un buffer a un flujo **FileInputStream**, de manera que se mejore la eficiencia de los accesos a los dispositivos en los que se almacena el fichero con el que conecta el flujo de manera automática, sin tener nosotros que preocuparnos por el mecanismo de buffering.

**EJEMPLO 5**: Como usar la clase **StreamTokenizer**, que obtiene un flujo de entrada y lo divide en "tokens". El flujo tokenizer puede reconocer identificadores, números y otras cadenas. El ejemplo muestra cómo utilizar la clase para contar números y palabras de un fichero de texto. Se abre el flujo de caracteres con ayuda de la clase **FileReader** y puedes ver cómo se "envuelve" (en amarillo) con el flujo **StreamTokenizer**.

```
package tokenizer;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;

public class Token {
```

```
    public void cuentaPalaNum(String fichero) {
       StreamTokenizer st = null;
       int np = 0, nn = 0;  // Nº de palabras y números
       int ta;  // Token actual
       try {
          st = new StreamTokenizer( new FileReader(fichero) );
          while( (ta = st.nextToken()) != StreamTokenizer.TT_EOF){
             if (st.ttype == StreamTokenizer.TT_WORD)
                np++;
             else if (st.ttype == StreamTokenizer.TT_NUMBER)
                nn++;
          }
          System.out.println("Palabras en el fichero: " + np);
          System.out.println("Números en el fichero: " + nn);
       } catch (FileNotFoundException e) {
          System.out.println( e.getMessage() );
       } catch (IOException e) {
          System.out.println( e.getMessage() );
       }
    }

    public static void main( String[] args ) {
        new Token().cuentaPalaNum( "c:\\datos.txt" );
    }
}
```

El método **nextToken()** devuelve un **int** que indica el tipo de token leído. Hay una serie de constantes definidas para tipos de token:

- **TT_WORD** indica que el token es una palabra.
- **TT_NUMBER** indica que el token es un número.
- **TT_EOL** indica que se ha leído el fin de línea.
- **TT_EOF** indica que se ha llegado al fin del flujo de entrada.
- **QUOTE_CHARACTER**: carácter entrecomillado.
- **DOUBLE_QUOTE_CHARACTER**: texto entrecomillado con comillas dobles.

En el código de la clase, apreciamos que se itera hasta llegar al fin del fichero. Para cada token, se mira su tipo y según el tipo se incrementa

el contador de palabras o de números.

## FLUJOS PREDEFINIDOS.

Tradicionalmente, los usuarios del sistema operativo Unix, Linux y también MS-DOS, han utilizado un tipo de entrada/salida conocida como **entrada/salida estándar**. El fichero de entrada estándar (**stdin**) es típicamente el teclado. El fichero de salida estándar (**stdout**) es típicamente la pantalla (o la ventana del terminal). El fichero de salida de error estándar (**stderr**) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.

Java tiene acceso a la entrada/salida estándar a través de la clase **System**. En concreto, los tres ficheros que se implementan son:

- **stdin**. Es un objeto de tipo InputStream, y está definido en la clase System.in como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.
- **stdout**. System.out implementa stdout como una instancia de la clase PrintStream. Se pueden utilizar los métodos **print()** y **println()** con cualquier tipo básico Java como argumento.
- **stderr**. Es un objeto de tipo PrintStream. Es un flujo de salida definido en la clase System y representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.

Para la entrada, se usa el método **read()** para leer que está sobrecargado:

- **int System.in.read();** Lee el siguiente byte (char) de la entrada

estándar.

- **int System.in.read(byte[] b);** Lee un conjunto de bytes de la entrada estándar y lo almacena en el array b.

Para la salida, se usa el método **print()** y similares para escribir:

- **System.out.print(String);** Muestra el texto en la consola.
- **System.out.println(String);** Muestra el texto en la consola y después efectúa un salto de línea.

Normalmente, para leer valores numéricos, lo que se hace es tomar el valor de la entrada estándar en forma de cadena y entonces usar métodos que permiten transformar el texto a números (int, float, double, etc.) según se requiera como Integer.parseInt(), etc.

| Funciones de conversión. | |
|---|---|
| **Método** | **Funcionamiento** |
| `byte Byte.parseByte(String)` | Convierte una cadena en un número entero de un byte |
| `short Short.parseShort(String)` | Convierte una cadena en un número entero corto |
| `intInteger.parseInt(String)` | Convierte una cadena en un número entero |
| `longLong.parseLong(String)` | Convierte una cadena en un número entero largo |
| `floatFloat.parseInt(String)` | Convierte una cadena en un número real simple |
| `doubleDouble.parseInt(String)` | Convierte una cadena en un número real doble |
| `booleanBoolean.parseBoolean(String)` | Convierte una cadena en un valor lógico |

*Figura 4: Funciones de conversión a tipos primitivos.*

**EJEMPLO 6**. Veamos un ejemplo en el que se lee por teclado hasta pulsar la tecla de retorno, en ese momento el programa acabará imprimiendo por la salida estándar la cadena leída.

> **Nota**: *Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase* ***StringBuffer*** *o* ***StringBuilder****. La primera permite almacenar cadenas que cambian durante la ejecución*

*del programa. StringBuilder es similar, pero no es síncrona. De este modo, para la mayoría de las aplicaciones, donde se ejecuta un solo hilo, supone una mejora de rendimiento sobre StringBuffer, que es mejor para aplicaciones multihilo. El proceso de lectura ha de estar en un bloque* ***try-catch****.*

```
import java.io.IOException;

public class LeeEstandar {
   public static void main(String[] args) {
      // Cadena donde ir almacenando los caracteres
      StringBuilder str = new StringBuilder();
      char c;
      // Por si ocurre una excepción usamos el bloque try-cath
      try {
         // Mientras la entrada de teclado distinta de Intro
         while ( (c=(char)System.in.read()) != '\n' ) {
            str.append(c); // Añadir el caracter leído a str
         }
      } catch( IOException e) {
         System.out.println( e.getMessage() );
      }
      // Escribir la cadena que se ha ido tecleando
      System.out.println("Cadena introducida: " + str);
   }
}
```

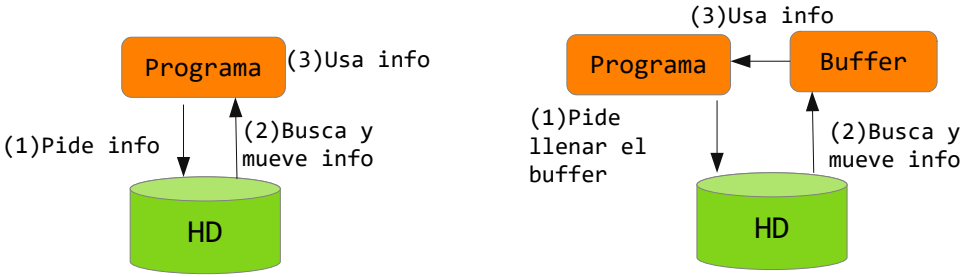
***RECUERDA****: Existen dos mecanismos básicos para gestionar los errores que se pueden producir en un bloque de código: el primero es utilizar la estructura* ***try-catch****, que permite intentar capturar un error que se pueda producir. Se pone el try, el código en cuestión y se finaliza con el catch. En esa sección del catch es donde pondremos las operaciones a realizar cuando se detecte un error. Se pueden poner varios bloques catch: una genérica y otras más detalladas. El otro mecanismo consiste en declarar que nuestro método puede lanzar una excepción (o varias), pero no gestionarla en ese método sino propagarla al que llame al método*

*(que tiene de nuevo esas dos posibilidades, y así hasta el método main()). En ese caso usamos throws lista_excepciones en la declaración del método.*

### 7.1.2. MEJORAR EL RENDIMIENTO CON BUFFERS.

La entrada y salida de datos a disco es un cuello de botella para los programas, puesto que la velocidad de trabajo de los discos duros (orden de milisegundos) en comparación con la velocidad de la RAM (orden de nanosegundos) es más lenta.

Para reducir esta diferencia, se intenta reducir la cantidad de operaciones de E/S que realiza el programa. Una forma de conseguirlo es usando buffers. Un buffer es una memoria temporal que se utiliza como almacén intermediario entre los discos y el programa. De esta forma, el programa pide al disco que le llene el buffer y va consumiendo la información desde el buffer.




**a) Sin buffers (mucha E/S -> lento)** **b) Con buffers -E/S -> +Rendimiento**

## TRABAJAR CON FICHEROS SIN BUFFERS

En primer lugar vamos por ejemplo a contar los bytes a cero de un fichero y vamos a obtener estos bytes mediante un stream binario de bajo nivel al que le iremos pidiendo los bytes de uno en uno.

**EJEMPLO 7**: leer fichero byte a byte sin usar buffers.

```
public class Ejem07 {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Fichero a usar: ");
      String nombre = sc.nextLine();
      FileInputStream entrada = null;
      long contador = 0;
      int dato;
      long comienza = System.currentTimeMillis(), acaba;
      try {
         entrada = new FileInputStream(nombre);
         while( (dato = entrada.read()) != -1) {
            if(dato == 0) contador++; // El trabajo que hace con el fichero
         }
         acaba = System.currentTimeMillis()
         System.out.print("El fichero tiene " + contador + " bytes a cero.");
         System.out.print("Segundos: " + (acaba - comienza)/1000.0);
   } catch(IOException e) {
      System.out.println( e.getMessage() );
   } finally { // Cerrar los ficheros
      entrada.close();
   }
}
```

Si ejecutamos el programa y usamos un fichero que tenga 1MByte o más de peso, veremos que el tiempo empleado es significativo. El motivo es que necesita hacer más de 1 millón de operaciones de E/S (en cada una lee 1 byte).

## USAR BUFFERS DE FORMA MANUAL

Repetimos el código anterior pero ahora definimos una zona de memoria para usarla como buffer. En vez de leer byte a byte, pedimos al stream que nos llene el buffer. A veces podrá hacerlo, y a veces no lo llenará comlpetamente, pero nos indicará siempre cuantos datos ha dejado en el buffer. Cuando no queden datos por mover devueve -1.

**EJEMPLO 8**: leer fichero usando buffer.

```
public class Ejem08 {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Fichero a usar: ");
```

```
        String nombre = sc.nextLine();
        FileInputStream entrada = null;
        long contador = 0;
        int leidos = 0;
        byte[] buffer = new byte[1024]; // buffer de 1KB
        long comienza = System.currentTimeMillis(), acaba;
        try {
           entrada = new FileInputStream(nombre);
           while( (leidos = entrada.read(buffer)) != -1) {
              // Ahora los datos están en buffer en posiciones 0..leidos-1
              for(int i= 0; i < leidos; i++)
                 if(buffer[i] == 0) contador++; // Ahora uso la info de buffer
           }
           acaba = System.currentTimeMillis()
           System.out.print("El fichero tiene " + contador + " bytes a cero.");
           System.out.print("Segundos: " + (acaba - comienza)/1000.0);
     } catch(IOException e) {
        System.out.println( e.getMessage() );
     } finally { // Cerrar los ficheros
        entrada.close();
     }
  }
```

Si vuelves a jecutar el programa contra el mismo fichero grande, verás que el tiempo empleado se reduce. Puedes probar a aumentar el tamaño del buffer (2KB, 32KB, 64KB) y verás que cuanto mayor sea, menos tiempo necesita para trabajar. Sin embargo, a partir de cierto tamaño el consumo de memoria no compensa la mejora de velocidad.
Otro inconveniente de usar manualmente los buffers, es que tu código se complica y no solo teienes que preocuparte de realizar el trabajo que querías, sino también de gestionar los buffers (que se acabe la información y haya que recargarlo de nuevo, etc.).

## USAR BUFFERS DE FORMA AUTOMÁTICA

Repetimos el código anterior pero ahora usamos un stream que ya sepa manejar buffers, de esta manera nosotros nos liberamos de estas preocupaciones y nos centramos en la lógica de nuestro problema a resolver, simplificando la programación.

**EJEMPLO 9**: leer fichero usando buffer de manera transparente.

```
public class Ejem08 {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Fichero a usar: ");
      String nombre = sc.nextLine();
      FileInputStream entrada = null;
      BufferedInputStream entradaConBuffer = null;
      long contador = 0;
      int dato = 0;
      long comienza = System.currentTimeMillis(), acaba;
      try {
         entrada = new FileInputStream(nombre);
         entradaConBuffer = new BufferedInputStream(entrada, 1024);
         while( (dato = entradaConBuffer.read()) != -1) {
               if(dato == 0) contador++; // Parece que no, pero uso buffers
         }
         acaba = System.currentTimeMillis()
         System.out.print("El fichero tiene " + contador + " bytes a cero.");
         System.out.print("Segundos: " + (acaba - comienza)/1000.0);
   } catch(IOException e) {
      System.out.println( e.getMessage() );
   } finally { // Cerrar los ficheros
      entradaConBuffer.close();
   }
}
```

El uso de buffers es casi transparente. Tener en cuenta simplemente que si usamos buffers en streams de salida (como `BufferedOutputStream`), antes de cerrar deberíamos indicarles que vuelquen contenido pendiente de ir al fichero con: `stream.flush()`.

## 7.1.3. LAS CLASES CONSOLE Y SCANNER

La entrada y salida de datos en las primeras etapas de Java se realizaba a través de los flujos predefinidos System.in y System.out.

**EJEMPLO 10**: lectura de datos de forma primitiva.

```
import java.io.*;

public class Ejemplo6 {

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

```
            BufferedReader br = new BufferedReader(
                                        new InputStreamReader( System.in ) );
            String entrada = br.readLine();
            System.out.println( "Has tecleado: " + entrada );
        }
    }
```

## LA CLASE CONSOLE

Los objetos **System.in** y **System.out** son flujos disponibles desde las primeras versiones de Java. En Java 6, se introdujo la clase **java.io.Console** para mejorar la entrada y salida de datos.

La clase **Console es un singleton** y como máximo puede haber un único objeto disponible en una JVM. Lo crea automáticamente la JVM y es accesible llamando al método **System.console()** que puede devolver null si el entorno donde se ejecuta el programa no tiene una consola para interactuar con el usuario.

**EJEMPLO 11**: El mismo ejemplo anterior con Console.

```
public class Ejemplo {

    public static void main(String[] args) {
        Console c = System.console();
        if( c != null) {
            String entrada = c.readLine();
            c.writer().println("Ha tecleado: " + entrada);
        }
    }
}
```

### Métodos reader() y writer() de Console

La clase Console ofrece acceso a una instancia de Reader y PrintWriter usando estos métodos. Acceder a estas clases es equivalente a llamar a System.in y System.out directamente, aunque con mejoras para trabajar con texto al usar Reader/Writer en vez de InputStream/OutputStream. Así, la codificación de caracteres se automatiza y se trabaja mejor con Strings.

## Métodos format() y printf() de writer()

Para salida de texto puedes usar estos dos métodos de writer(). Son equivalentes y puedes definir campos. Por ejemplo, con %$n puedes usar la variable n, varias veces si quieres.

**EJEMPLO 12:** imprimir información para el usuario.

```
public class Salida {

    public static void main(String[] args)
    throws NumberFormatException, IOException {
        Console c = System.console();
        if( c == null ) {
            throw new RuntimeException("No hay Consola");
        } else {
            c.writer().println("Bienvenido al Zoo!");
            c.format("Nuestro zoo tiene 491 animales y 25 empleados.");
            c.writer().println();
            c.printf("Tiene una extensión de %d Km2 y %$1 es mucho", 10);
        }
    }
}
```

## Método flush() de Console

Fuerza a que cualquier mensaje almacenado en los buffers sea escrito de forma inmediata. Es conveniente llamarlo antes de usar readLine() o readPassword().

## Método readLine()

El método recupera una línea de texto (hasta que el usuario pulsa enter). También tiene una variante sobrecargada **readLine(String format, Object. . . args)**, que muestra un prompt al usuario antes de mostrar el texto.

**EJEMPLO 13**: usar varias formas de leer y escribir datos con Console:

```
public class LeerDatos {

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

```
        throws NumberFormatException, IOException {
            Console c = System.console();
            if( c == null) {
                throw new RuntimeException("Consola no disponible");
            } else {
                c.writer().print("Como ha ido el dia? ");
                c.flush();
                String dia = c.readLine();
                String nombre = c.readLine("Su nombre: ");
                Integer edad = null;
                c.writer().print("Su edad? ");
                c.flush();
                BufferedReader br = new BufferedReader( c.reader() );
                String valor = br.readLine();
                edad = Integer.valueOf(valor);
                c.writer().println();
                c.format("Se llama " + nombre );
                c.writer().println();
                c.format("Su edad es " + edad);
                c.printf("Su dia ha ido: " + dia );
            }
        }
    }
```

### Método `readPassword()` de Console

Similar a `readLine()` pero no muestra lo que se teclea. Tiene otra versión sobrecargada `readPassword(String format, Object...args)`. Por motivos de seguridad devuelve un array de caracteres en vez de un String, debido a que los String van a una zona de memoria de largo uso y si hay un volcado de memoria por errores, podrían recuperarse los passwords tecleados que permanezcan en esta zona (un hacker incluso podría hacer programas que provocasen esta situación) y copiar luego el archivo de volcado.

**EJEMPLO 14**: pedir y comprobar un password.

```
    public class PedirPassword {
        public static void main(String[] args)
        throws NumberFormatException, IOException {
            Console c = System.console();
            if( c == null) {
                throw new RuntimeException("Consola no existe");
            } else {
```

```
                char[] pwd = c.readPassword("Password: ");
                c.format("Teclee de nuevo: ");
                c.flush();
                char[] verifica = c.readPassword();
                boolean ok = Arrays.equals(pwd, verifica);
                // Eliminar los passwords de memoria
                Arrays.fill(pwd, 'x');
                Arrays.fill(verifica, 'x');
                c.format("Su password es " + (ok? "correcto": "incorrecto") );
            }
        }
    }
```

## LA CLASE SCANNER

La clase **Scanner** se introdujo para hacer sencilla la lectura de tipos básicos desde un origen con datos en formato carácter. La clase Scanner está en el paquete **java.util**. Para usarla necesitas crear un objeto Scanner. El constructor indica el origen del que leer y actúa como una envoltura. El origen puede ser un `Reader`, un `InputStream`, un `String`, o un `File`. Por ejemplo puedes usarlo desde la entrada estándar como ya hemos estado haciendo:

```
Scanner entradaStandar = new Scanner( System.in );
```

Si origenChar es un `Reader`, puedes crear un Scanner:

```
Scanner scanner = new Scanner( origenChar );
```

Al procesar la entrada, el scanner trabaja con tokens (un trozo de string de caracteres que no puede dividirse en trozos más pequeños sin dejar de representar un elemento de información). Los tokens deben estar separados por delimitadores, que por defecto son espacios en blanco, tabuladores, salto de línea. Puedes cambiar los delimitadores si lo necesitas. Si s es una variable de tipo `Scanner`, tiene los siguientes métodos:

- **s.next()** — lee el siguiente token (palabra) y lo devuelve en un String.

- **s.nextInt()**, **s.nextDouble()**, etc. — lee el siguiente token e intenta convertirlo al tipo indicado. Hay métodos para cada tipo primitivo.
- **s.nextLine()** — lee una línea completa hasta el final de la misma y devuelve un String. El final de la línea no forma parte del valor devuelto.

Todos estos métodos pueden generar excepciones. Si intentas leer un dato cuando has pasado el final, se genera una excepción de tipo **NoSuchElementException**. Y un método como s.nextInt() puede lanzar una excepción de tipo **InputMismatchException** si el token de la entrada no representa un valor del tipo indicado (entero en este caso). Las excepciones no necesitan tratramiento forzoso.

La clase Scanner te permite saber si hay más tokens en la entrada, o si son del tipo que esperas. Eso ayuda a controlar errores sin recurrir a excepciones:

- **s.hasNext()** — devuelve **true** si al menos hay otro token en la entrada.
- **s.hasNextInt()**, **s.hasNextDouble()**, etc — devuelve **true** si hay al menos un token que representa un dato del tipo indicado.
- **s.hasNextLine()** — devuelve true si hay al menos una línea más en la entrada.

Con tantas clases para leer caracteres (BufferedReader, TextReader, Scanner) podrías tener dudas sobre cuál usar. Utiliza Scanner salvo que tengas alguna razón para no hacerlo. Ten en cuenta además que puedes cambiar el delimitador que usa, incluso puedes indicar los delimitadores con expresiones regulares.

## 7.1.4. E/S DE TIPOS DE DATOS Y DE OBJETOS.

### TIPOS PRIMITIVOS

Las clases **PrintWriter**, **Scanner**, **DataInputStream** y **DataOutputStream** te permiten hacer E/S de tipos primitivos de forma sencilla, ya sea en formato de texto o en formato binario. Para formato de texto:

- **PrintWriter**: print(), println() y printf() sobrecargados para cada tipo de dato primitivo y Strings, genera salida de texto.
- **Scanner**: permite leer tipos de datos (ya lo conocemos) a partir de texto.

Para mover tipos de datos primitivos (boolean, char, byte, short, int, long, float y double) y Strings en formato binario podemos usar los streams de datos que implementan las interfaces DataInput y DataOutput. Las clases más genéricas que las implementan son: **DataInputStream** y **DataOutputStream**.

Algunos métodos de **DataOutputStream**:

- **void write(int b):** escribe el byte almacenado en el entero b.
- **void write(byte[] b)**: escribe los b.length bytes del array b.
- **void write(byte[] b, int desp, int n)**: escribe n bytes del array b comenzando por el desp (desplazamiento).
- **writeByte(int b)**: escribe el byte b.
- **writeBoolean(boolean b)**: escribe el booleano b.
- **writeChar(char c)**: escribe el char c (2 bytes).
- **writeShort(short s)**: escribe el short.
- **writeInt(int i)**: escribe entero.
- **writeLong(long i)**: escribe un long.
- **writeFloat(float f)**: escribe un float.
- **writeDouble(double d)**: escribe un double.

- `writeUTF(String s)`: escribe lee los bytes de un String.
- `long skip(long n)` y `int skipBytes(int n)`: intentan saltar n bytes del stream, devuelve los que efectivamente han saltado.

Algunos métodos de `DataInputStream`:

- `int read()`: devuelve el siguiente byte del stream almacenado en un int para tener valores 0-255.
- `int read(byte[] b)`: lee hasta b.length bytes del stream y los almacena en el array b. Devuelve cuantos ha leído.
- `byte readByte()`: lee el siguiente byte y lo devuelve en un byte.
- `boolean readBoolean()`: lee el byte de un boolean y lo devuelve.
- `char readChar()`: lee los dos siguientes bytes del stream y devuelve el char que representan.
- `short readShort()`: lee los dos siguientes bytes del stream y devuelve un short con signo.
- `int readInt()`: lee 4 bytes del stream y devuelve un int.
- `long readLong()`: lee 8 bytes del stream y devuelve un long.
- `float readFloat()`: lee 4 bytes del stream y devuelve un float.
- `double readDouble()`: lee 8 bytes del stream y devuelve un double.
- `String readUTF()`: lee los bytes de un String en caracteres UTF.
- `long skip(long n)` y `int skipBytes(int n)`: intentan saltar n bytes del stream y devuelve los que efectivamente han posido saltar.

**EJEMPLO 15**: Crear un fichero binario llamado personas.dat usando un DataOutputStream, que contenga registros (un registro es equivalente a una fila de una tabla de base de datos) con los campos siguientes que representan información de una persona:

nºRegistro + Nombre + edad + esSoltero + estatura.

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

```java
public class Ejemplo15 {

    public static void main(String[] args) {
        String nombre;
        int edad, contador = 0;
        double estatura;
        boolean soltera;
        try (
            DataOutputStream dos = new DataOutputStream(
                                   new FileOutputStream("personas.dat") );
            Scanner sc = new Scanner(System.in);
        ) {
            do {
                System.out.print("Nombre (ENTER para acabar): ");
                nombre = sc.nextLine();
                if(!nombre.isEmpty()) {
                    System.out.print("edad (entero): ");
                    while(!sc.hasNextInt() ) sc.nextLine();
                    edad = sc.nextInt();
                    System.out.print("Soltero (true/false): ");
                    while(!sc.hasNextBoolean() ) sc.nextLine();
                    soltera = sc.nextBoolean();
                    System.out.print("Estatura (metros): ");
                    while(!sc.hasNextDouble() ) sc.nextLine();
                    estatura = sc.nextDouble();
                    // Guardar el registro
                    dos.writeInt(contador++);
                    dos.writeUTF(nombre);
                    dos.writeInt(edad);
                    dos.writeBoolean(soltera);
                    dos.writeDouble(estatura);
                    sc.nextLine(); // vaciar escanner
                }
            }while( !nombre.isEmpty() );
        }catch(IOException ioe) {
             System.out.print( ioe.getMessage() );
        } // try-con-recursos-a-cerrar
    } // main()
} // clase
```

**EJERCICIO 2**: Ejecuta el ejemplo 15 y crea dos o tres personas, luego haz un programa que recorra el fichero y muestre los registros por

consola usando un stream DataInputStream.

## E/S DE OBJETOS

Pero, ¿qué ocurre si quieres leer/escribir un objeto a través de un stream? Tradicionalmente tenías que inventar una forma de codificar tus objetos como una secuencia de datos de tipos primitivos expresados como caracteres o en binario. Este proceso de codificación se llama **serializar el objeto**. Al realizar la lectura de un objeto serializado, tienes que leer esos datos serializados y reconstruir una copia del objeto. Para objetos complejos esta tarea puede ser muy complicada.

También puedes pasarle el trabajo a Java si utilizas las clases **ObjectInputStream** y **ObjectOutputStream**. Son subclases de **InputStream** y de **OutputStream** respectivamente.

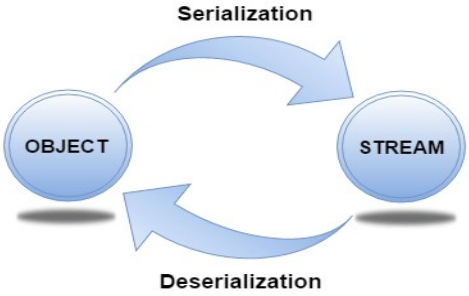



*Figura 5: Concepto de serializar y deserializar.*

**ObjectInputStream** y **ObjectOutputStream** son envolturas de sus dos subclases. Esto hace que sea posible realizar E/S de objetos usando cualquier stream de bytes. Los métodos más destacados que tiene son **`Object readObject()`** de `ObjectInputStream` y **`writeObject(Object obj)`** de `ObjectOutputStream`. Ambos pueden lanzar excepciones de tipo `IOException`. Observa que `readObject()` devuelve un valor de tipo `Object`, al

que hay que aplicar un casting.

ObjectOutputStream tiene los métodos **writeInt(), writeDouble(), etc.** que tiene un DataOutputStream, para salida de datos primitivos y ObjectInputStream tiene sus correspondientes métodos de lectura de estos tipos primitivos: **readInt(), readDouble(), etc.**

Como los flujos de objetos son streams de bytes, los objetos se representan de forma binaria (esto es bueno por eficiencia) pero causa problemas si cambias al objeto de entorno o de lenguaje de programación (un programa Java lo escribe y un programa en C# o C++ lo lee, o al contrario). Por este motivo, estas clases tienen capacidades de almacenamiento de objetos limitadas, no sirven para enviar objetos por la red o comunicarse con otros programas que no sean Java. En estos últimos casos se utilizan otras aproximaciones para serializar objetos.

ObjectInputStream y ObjectOutputStream solamente trabajan con objetos que implementen la interface **Serializable**.

Cuando se escriben objetos a un fichero con estos streams, si el mismo objeto aparece varias veces dentro de otro, solo se guarda una vez, la primera aparición y se guardan referencias posteriormente. Si desde que se guarda una primera versión el objeto, este objeto sufre cambios en memoria, y una vez cambiado se vuelve a almacdenar, estos cambios se perderán.

> ***Importante:*** *no puedes usar un fichero para almacenar un objeto, luego hacer cambios al objeto y volverlo a almacenar. Si lo haces, no se guardan los cambios. Tras cambiar el objeto, debes recrear el fichero cerrando el stream y repetir el proceso de generarlo.*

Ten en cuenta además que cuando se deserializa un objeto no se llama a los constructores del objeto ni se ejecuta código inicializador alguno.

Las interfaces y clases relacionadas con la serialización:

- **Serializable**: Solamente los objetos que implementan esta interface pueden ser serializados usando las librerías core de Java. `Serializable` no define métodos abstractos, solo se utiliza para indicar que el objeto puede ser serializado. Las variables estáticas no se almacenan. Tampoco variables con el modificador **trasient**.
- **Externalizable**: Aunque Java intenta automatizar el proceso de serializar/deserializar objetos, hay casos en que los programadores necesitan controlar este proceso (usar compresión de datos con mucho peso, criptografía para datos sensibles, etc.). Para estas situaciones se usa esta interface que define dos métodos:
  - `void readExternal(ObjectInput inStream) throws IOException, ClassNotFoundException`
  - `void writeExternal(ObjectOutput outStream) throws IOException`
- **ObjectOutput**: Extiende la interface `DataOutput` y soporta la serialización de objetos. Define métodos como **writeObject()**. Todos pueden lanzar excepciones **IOException**.
- **ObjectOutputStream**: esta clase extiende a `OutputStream` e implementa la interface `ObjectOutput`. Es responsable de escribir objetos a un stream. Su constructor debe indicar el flujo.
  - `ObjectOutputStream(OutputStream outStream) throws IOException`

  Tiene una clase interna llamada **PutField** que permite escribir miembros persistentes (se sale de nuestro presupuesto, lo comento por si os interesa mirarlo).

- **ObjectInputStream**: extiende la clase InputStream e implementa la interface ObjectInput. Es responsable de leer objetos desde un flujo. El constructor indica el flujo de entrada del que lee, es:
  - `Object InputStream(InputStream inStream) throws IOException`

  Tiene una clase interna llamada **GetField** para leer campos estáticos.

## MÉTODOS DE ObjectOutputStream

- **void close()** Cierra el stream. Si intentas escribir después se genera una IOException.
- **void flush()** Los buffers usados se llevan al dispositivo de salida y se vacían. Es importante llamarlo antes de cerrar el flujo para asegurar que toda la información se vuelca en el destino.
- **Todos los métodos de escritura de tipos primitivos** presentes en DataOutputStream del tipo writeXX(): (writeInt(), ...).

## MÉTODOS DE ObjectInputStream

- **int available()**: nº de bytes disponibles en el buffer de entrada.
- **void close()** Cierra el flujo. Un intento posterior de leerlo lanza la excepción IOException.
- Todos los readXX() de tipos primitivos: read(), readInt(), ...

**EJEMPLO 16: SERIALIZAR/DESERIALIZAR.** Instancia la clase UnaClase. El objeto tiene 3 variables de instancia de tipos String, int y double. Esta es la información a guardar y restaurar.

```
public class Ejemplo16 {

   public static void main(String args[]) {
      // serializacion
      try {
         UnaClase ob1 = new UnaClase("Hola", -7, 2.7e10);
         System.out.println("ob1: " + ob1);
         FileOutputStream fos = new FileOutputStream("serial.bin");
```

```
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ob1);
            oos.flush();
            oos.close();
         } catch(IOException e) {
            System.out.println("Fallo al serializar: " + e);
            System.exit(0);
         }
         // deserializar
         try {
            UnaClase ob2;
            FileInputStream fis = new FileInputStream("serial.bin");
            ObjectInputStream ois = new ObjectInputStream(fis);
            ob2 = (UnaClase)ois.readObject();
            ois.close();
            System.out.println("ob2: " + ob2);
         } catch(Exception e) {
            System.out.println("Fallo al deserializar: " + e);
            System.exit(0);
         }
      }
   }

   class UnaClase implements Serializable {
      String s;
      int i;
      double d;
      public UnaClase(String s, int i, double d) {
         this.s = s;
         this.i = i;
         this.d = d;
      }

      public String toString() {
         return String.format("UnaClase={s:%s, i:%d, d:%8.2g}", s,i,d);
      }
   }
```

Su salida:

```
ob1: UnaClase={s:Hola, i:-7, d:2.7E10}
ob2: UnaClase={s:Hola, i:-7, d=2.7E10}
```

Lo bueno de esta forma de trabajar es que incluso serializar y deserializar objetos complejos como un árbol es igual de sencillo, se copian los enlaces, los objetos referenciados, los objetos anidados (también deben ser serializables para que ocurra), etc.

**EJERCICIO 3**. Usa el código del ejemplo 16. Cuando serialices el objeto, haz un cambio en alguna de sus variables de instancia y vuelve a enviarlo por el string. Luego, al deserializar recupera los dos objetos serializados y los imprimes. ¿Qué ocurre?

**EJERCICIO 4**. Si una superclase es serializable, ¿Lo serán sus clases hijas? Usa el código del ejemplo 16 para crear una subclase de la clase `UnaClase`, instancia un objeto suyo y lo serializas y deserializas. ¿Funciona?

**EJERCICIO 5**. Si una clase usa composición ¿es serializable si lo son sus partes? Es decir, si tenemos estructuras complejas, llamando a serializar el objeto, ¿el proceso se realiza bien?

a) Copia el código del ejemplo 16. Crea una clase llamada `UnaParte` que tenga dos variables miembro: un `String` llamado `usuario` y otro `String` llamado `password` modificada como `trasient`. Añade a la clase `UnaClase` una variable miembro de tipo `UnaParte` y en el constructor la instancias con los valores que quieras. Modifica el método `toString()` para que puedas ver el valor serializado y deserializado. Ejecuta el programa ¿Qué ocurre?

b) Marca la clase `UnaParte` como `Serializable` y repite la ejecución ¿Funciona? ¿Pueden serializarse objetos complejos?

## 7.2 FICHEROS.

Los datos (incluidos los programas) almacenados en la memoria RAM de un ordenador sobreviven mientras se alimente de electricidad al

hardware. Si quieres que permanezcan más tiempo (hacerlos más persistentes) debes almacenarlos en unidades de memoria secundaria (discos duros, memorias USB, trarjetas SD, CD's, DVD's, etc.). Cuando se almacenan en estos dispositivos, lo hacen en forma de ficheros, que son colecciones de datos. Los ficheros tienen un nombre (entre otras cosas más) y se organizan dentro del dispositivo en carpetas o directorios. Un directorio puede contener ficheros y otros directorios.

Los programas pueden leer datos de ficheros existentes y pueden crear nuevos ficheros donde escribirlos. En Java, los ficheros que contienen datos en formato entendible por humanos pueden leerse usando objetos de la clase **FileReader** (subclase de Reader) y escribirse usando objetos **FileWriter** (subclase de Writer). Para ficheros que almacenan los datos en binario se usan otros streams como **FileInputStream** y **FileOutputStream**. Todas estas clases están en el paquete **java.io**.

## 7.2.1 LEER Y ESCRIBIR FICHEROS.

Los pasos fundamentales para manipular ficheros son:

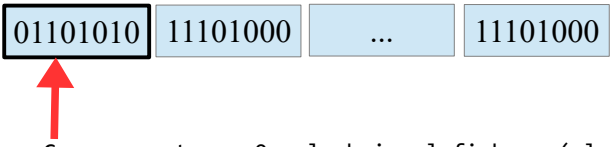
- Abrirlo.
- Trabajar con él
- Cerrarlo.

Además de esas consideraciones, debemos tener en cuenta también las clases Java a emplear, es decir, recuerda que hemos comentado que si vamos a tratar con ficheros de texto, hay que usar FileReader y FileWriter y si queremos bytes debemos usar FileInputStream y FileOutputStream.

Otra cosa a considerar, cuando se va a hacer uso de ficheros, es la

forma de acceso al fichero que se va a utilizar: si va a ser de **manera secuencial** o bien **aleatoria**. En un fichero secuencial, para acceder a un dato debemos recorrer todo el fichero desde el principio hasta llegar a su posición.

Puedes imaginar el fichero como un array de bytes pero almacenados en el disco. El fichero usa internamente una marca llamada **puntero** o **cursor** que le indica la posición donde se va a leer/escribir datos. Al abrir el fichero para lectura, esa marca está en la primera posición que es la cero. Cuando lees un byte, el puntero se mueve al siguiente byte. Ocurre igual si escribes en el fichero.




Cursor=puntero= 0 al abrir el fichero (el primer byte)

Cuando lees un byte, el puntero avanza al siguiente, si lo hay. Si para llegar al cuarto byte, debes leer los 3 anteriores (no accedes de manera directa a cualquier lugar) estás usando acceso secuencial.

Sin embargo, en un fichero utilizado con acceso aleatorio tienes la posibilidad de acceder al elemento que ocupa una posición arbitraria, porque puedes cambiar la posición del puntero para que apunte a cualquier lugar del fichero y ahí leer o escribir.

Si un fichero tiene acceso aleatorio también tendrá acceso secuencial. Pero lo contrario no es cierto. Tened en cuenta también que el hecho de que un stream permita acceso aleatorio no significa que sea posible utilizarlo con cualquier fichero, porque eso dependerá también de como los programas que han creado el fichero hayan organizado la información que contiene en su interior.

## FICHEROS DE TEXTO Y BINARIOS

Ya comentamos que los ficheros se utilizan para guardar la información en un soporte: disco duro, disquetes, memorias usb, dvd, etc., y posteriormente poder recuperarla. También distinguimos dos tipos de ficheros: los de texto y los binarios. En los ficheros de texto la información se guarda como caracteres. Esos caracteres están codificados en Unicode, o en ASCII u otras codificaciones de texto. En la siguiente porción de código puedes ver cómo para un fichero existente, que en este caso se llama texto.txt, averiguamos la codificación que posee, usando el método **getEncoding()**

```
FileInputStream fichero;
try {
   // Elegir fichero para leer flujos de bytes "crudos"
   fichero = new FileInputStream("c:/texto.txt");
   // InputStreamReader es un puente de bytes a caracteres
   InputStreamReader isr = new InputStreamReader(fichero);
   // Vemos la codificación actual
   System.out.println( isr.getEncoding() );
}
catch(IOException ex) {
  System.out.println( ex.getMessage() );
}
```

Para archivos de texto, se puede abrir el fichero para leer usando la clase **FileReader**. Esta clase nos proporciona métodos para leer caracteres. Cuando nos interese no leer carácter a carácter, sino leer líneas completas, podemos usar la clase **BufferedReader** envolviendo su **FileReader**. Lo podemos hacer de la siguiente forma:

```
File archivo = new File("C:/U07/fichero.txt");  // el archivo
FileReader fr = new FileReader(archivo);        // lee char a char
// Cambiamos a otro que además de char lee líneas, usa buffers...
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para escribir en archivos de texto lo podríamos hacer usando:

```
FileWriter fichero = new FileWriter("c:/U07/fichero.txt");
// Cambiamos a otro que además de char escribe tipos->char, ...
PrintWriter pw = new PrintWriter(fichero);
pw.println("Linea de texto");
pw.println(edad);
...
```

Si el fichero en el que queremos escribir existe y lo abrimos con FileWriter (para escribir en él) como en el ejemplo anterior, el contenido previo se pierde y se sustituye por el nuevo que mandemos por el stream. Si lo que queremos es añadir información sin destruir la que ya tenía, entonces usaremos otro constructor al que pasamos un segundo parámetro como true indicando que queremos añadir (append):

```
FileWriter("c:/U07/fichero.txt",true);
```

Los ficheros binarios por contra almacenan la información en bytes, la información está codificada en binario, pudiendo ser de cualquier tipo: fotografías, números, letras, archivos ejecutables, etc. Los archivos binarios guardan una representación de los datos en el fichero.

## MODOS DE ACCESO Y REGISTROS DE DATOS.

En Java no se impone una estructura en un fichero, por lo que conceptos como el de registro que si existen en otros lenguajes, en principio no existen en Java (sí en versiones más recientes del lenguaje). Por tanto, los programadores deben organizar la información que añaden a sus ficheros de modo que cumplan con los requerimientos de sus aplicaciones.

Si un programador necesita guardar datos en registros (no es muy normal porque tienes colecciones que se pueden serializar), el programador define sus registros de datos con el número de bytes que

le interesen, moviéndose luego por el fichero teniendo en cuenta ese tamaño que ha definido. Si cada registro puede tener un tamaño distinto al resto (**registros de tamaño variable**), esta navegación por el fichero de registros se complica (¿En qué posición del fichero comienza el registro 21? --> no se averigua de manera directa). Cuando comienza un registro podemos usar una marca, o guardar el tamaño de cada uno en otro lugar (a modo de directorio que los localice):

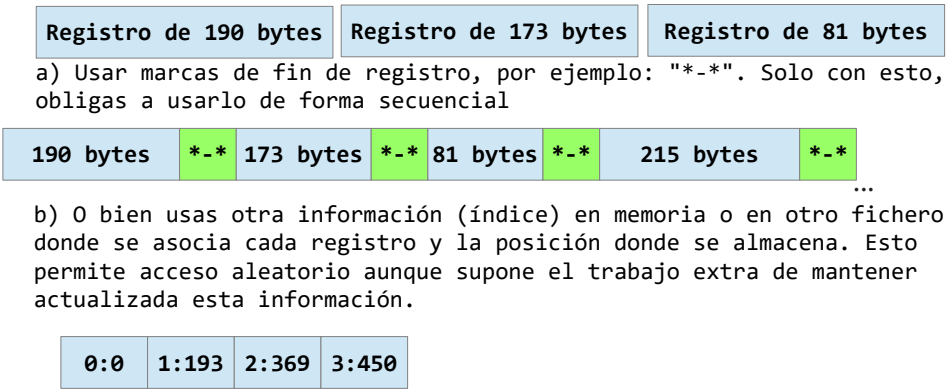



***Figura 7: Ficheros de registros de datos de tamaño variable.***

Si todos los registros en el fichero tienen la misma cantidad de bytes, digamos pesoRegistro, cambiar de un registro a otro en el fichero, o acceder a uno concreto es sencillo: ¿En qué posición del fichero comienza el registro 21? --> 21 * pesoRegistro.

Se dice que **un fichero es de acceso directo o aleatorio** o de **organización directa** cuando para acceder a un registro n cualquiera del fichero, no se tiene que pasar por los n-1 registros anteriores. En caso contrario, estamos hablando de **ficheros secuenciales**. Con Java se puede trabajar con ficheros secuenciales y con ficheros de acceso aleatorio.

En los ficheros secuenciales, la información se almacena de manera secuencial, por tanto, para recuperarla o procesarla (trabajar con ella) probablemente se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el tercer registro del fichero sin ayuda extra, en un fichero que almacena registros de longitud variable, tendremos que abrir el fichero y leer los primeros 2 registros, hasta que finalmente nos posicionemos en el registro número 3.

Por el contrario, si se tratara de un fichero de acceso aleatorio, podríamos acceder directamente a la posición donde comienza el registro 3 del fichero, o a cualquier otro que nos interese porque seremos capaces de calcular y acceder directamente a cualquier lugar del fichero donde comienza el registro.

**Operaciones con Acceso secuencial**.

En el siguiente ejemplo vemos cómo se escriben datos en un fichero secuencial: el nombre y apellidos de una persona utilizando el método **writeUTF()** que proporciona la clase **DataOutputStream**, seguido de su edad que la escribimos con el método **writeInt()** de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir leyendo de manera secuencial los datos almacenados en el fichero y escribiéndolos por la consola.

```
public class ModoSecuencial {
   public static void main(String[] args) {
     DataOutputStream fo = null;
     DataInputStream fi = null;
     String nombre = null;
     int edad = 0;
      try {
         // Crea o abre para añadir al archivo
         fo = new DataOutputStream(
```

```
                    new FileOutputStream("/U07/secuencial.dat", true)
                    );
            fo.writeUTF( "Antonio López Pérez" );
            fo.writeInt(33);
            fo.writeUTF( "Pedro Piqueras Peñaranda" );
            fo.writeInt(45);
            fo.writeUTF( "José Antonio Ruiz Pérez" );
            fo.writeInt(51);
            fo.close();              // Cerrar flujo
            // Abrir para leer
            fi = new DataInputStream(
                        new FileInputStream("/U07/secuencial.dat")
                 );
          nombre = fi.readUTF();  // Leer primer registro
          System.out.println(nombre);
          edad = fi.readInt();
          System.out.println(edad);
          nombre = fi.readUTF();  // Leer segundo registro
          System.out.println(nombre);
          edad = fi.readInt() ;
          System.out.println(edad);
          nombre = fi.readUTF();   // Leer tercer registro
          System.out.println(nombre);
          edad = fi.readInt();
          System.out.println(edad) ;
          fi.close();
        }
        catch(FileNotFoundException fnfe){ /*Archi.no encontrado*/ }
        catch (IOException ioe) { /* Error al escribir */ }
        catch (Exception e) { /* Error de otro tipo*/
            System.out.println( e.getMessage()); }
        }
    }
```

Observa el código, no ha escrito las cadenas de caracteres con el mismo tamaño y eso nos habría permitido saber con posterioridad el tamaño del registro que tenemos que leer. Cada registro tendrá un tamaño distinto, así que debemos leerlos de forma secuencial porque al no saber cuanto pesa cada registro, no sabemos donde comienza cualquiera de ellos salvo el primero que está en la posición cero del

fichero.

Para buscar un registro x en un fichero de registos de datos secuencial, tendremos que abrir el fichero e ir leyendo registros hasta encontrar el registro que buscamos.

**¿Y si queremos eliminar un registro en un fichero secuencial, qué hacemos?** Esta operación problemática, puesto que no podemos quitar el registro y tapar el hueco de manera eficiente. Una opción, aunque costosa, sería crear un nuevo fichero: recorremos el fichero original y vamos copiando registros en el nuevo hasta llegar al registro que queremos borrar. Ese no lo copiamos al nuevo fichero, y seguimos copiando el resto de registros hasta el final del fichero al nuevo fichero. De este modo, obtendríamos un nuevo fichero que sería el mismo que teníamos pero sin el registro que queríamos borrar. Pero claro si tienemos 400 registros, para borrar uno, tenemos que recorrer los 400 y crear un nuevo fichero donde copiar los 399 registros restantes. Una operación muy ineficiente. Por tanto, si sabemos que necesitamos borrar registros en el fichero, no es recomendable usar un fichero de tipo secuencial o bien usar borrados lógicos en vez de físicos (los comentamos más adelante).

La clase `Reader` (pero no todas sus subclases la soportan) define los métodos **`mark()`** y **`mark(int limite)`** que permiten marcar la posición actual del flujo para volver a esa posición al hacer un **`reset()`**. La marca se mantiene mientras no se lean **`limite`** cantidad de caracteres/bytes. Si no se soportan marcas, se lanza una excepción de tipo `IOException`. Se puede comprobar si puedes usar esta operación usando el método **`markSupported()`**.

**Acceso aleatorio**.

A veces no necesitamos leer un fichero de principio a fin, sino acceder al fichero como si el fichero fuera un array de registros, pero en disco en vez de en memoria RAM, donde se accede a un registro concreto del fichero. Java proporciona la clase **RandomAccessFile** para este tipo de entrada/salida. La clase **RandomAccessFile** permite utilizar un fichero de acceso aleatorio en el que el programador define el formato de sus registros.

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

Donde ruta es la dirección física en el sistema de archivos y modo:

- "r" para sólo lectura.
- "rw" para lectura y escritura.

La clase RandomAccessFile implementa las interfaces **DataInput** y **DataOutput**. Para abrir un archivo en modo lectura haríamos:

```
RandomAccessFile rafi = new RandomAccessFile("input.dat", "r");
```

Para abrirlo en modo lectura y escritura:

```
RandomAccessFile rafio = new RandomAccessFile("input.dat", "rw");
```

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases diferentes. Hay que especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura.

Dispone de métodos específicos de desplazamiento por el fichero como **seek()** y **skipBytes()** para poder mover el puntureo del fichero de un registro a otro, o posicionarse directamente en una posición concreta del fichero. Su utilización no está basada en el concepto de flujos.

**Para abrir un fichero con acceso aleatorio hay dos constructores:**

- Mediante el nombre del fichero:

```
f = new RandomAccessFile(String nombre, String modo);
```

- Mediante un objeto file:

```
f = new RandomAccessFile(File fichero, String modo);
```

El parámetro modo determina si se tiene acceso de sólo lectura (r) o bien de lectura/escritura (rw).

Para acceder a la información con un objeto `RandomAccessFile` se tiene acceso a todas las operaciones **read()** y **write()** de las clases **DataInputStream** y **DataOutputStream** y se dispone de muchos métodos para ubicarse dentro de un fichero:

- **long getFilePointer()** Devuelve la posición actual del puntero del fichero.
- **void seek(long pos)** Sitúa el puntero del fichero en una posición determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 es el comienzo de ese fichero.
- **long length()** Devuelve la longitud del fichero. La posición `length()` de un fichero indica el final del mismo.
- **int skipBytes(int desplazamiento)** Desplaza el puntero desde la posición actual, el número de bytes indicado por el desplazamiento. Si el desplazamiento es negativo o cero, no cambia el puntero del fichero. Si intentas desplazarte una cantidad de bytes superior a los bytes que hay hasta el final, se desplaza hasta el final. Devuelve la cantidad de bytes efectivamente desplazados.

Operaciones con ficheros `RandomAccessFile`:

- **Modificar información**: te posicionas en un lugar y la operación **write()** que realices, sobreescribe el contenido anterior.

- **Añadir nueva información**: te posicionas al final del fichero y vuelves a escribir.

```
// Añadir un nuevo registro de datos
raf = new RandomAccessFile("c:/U07/prueba.dat", "rw");
raf.seek( raf.length() );
// Escribir...
```

- **Borrar información existente**: sigue siendo una operación problemática. Para ganar eficiencia evitando tener que generar un nuevo fichero podemos usar un truco que consiste en cambiar el borrado físico por un borrado lógico. Para hacer un borrado lógico de registros se añade un campo booleano a cada registro que indique si está borrado o no. Así el borrado puede realizarse de manera lógica, no física. Para borrar un registro simplemente escribimos true en su campo de borrado. El inconveniente es que si hay muchos registros borrados en el fichero, gran parte del espacio en disco que consume el fichero no contiene información útil y además ralentiza los recorridos secuenciales que se haga del fichero. Para solucionar este inconveniente aparece una nueva operación denominada **pack** (empaquetar) que procesa el fichero y genera un fichero nuevo en el que se eliminan los registros borrados de manera lógica.

### 7.2.2 TRABAJAR CON FICHEROS Y DIRECTORIOS

Si solo utilizas el nombre del fichero para trabajar con él, se supone que está almacenado en el directorio actual (llamado también **directorio de trabajo** o **directorio por defecto**). El directorio actual puede cambiarlo el usuario o un programa. Si el fichero no está en el directorio actual, debes indicar la ruta que permita localizarlo (**path**).

Hay dos tipos de rutas (absolutas y relativas). Las relativas indican la trayectoria de directorios que hay que atravesar hasta llegar a donde

está el fichero y la absoluta comienzan en la raíz del sistema de ficheros hasta llegar a la carpeta donde se encuentra almacenado el fichero.

Además, la sintaxis de cada sistema de ficheros varía de un sistema operativo a otro. Ejemplos:

- dato.dat — un fichero de nombre "dato.dat" en el directorio actual.
- /home/eck/java/examples/dato.dat — ruta absoluta en UNIX/Linux y MacOS X al fichero de nombre dato.dat que está en un directorio llamado examples, dentro de un directorio llamado java, dentro a su vez de otro directorio llamado eck que a su vez está dentro del directorio home que cuelga de la raíz del sistema de ficheros y por eso es una ruta absoluta, porque aparece la raíz del sistema de ficheros como lugar del que parte la ruta.
- C:\\eck\\java\\examples\\datos.dat — ruta absoluta equivalente pero en sistemas operativos Windows.
- examples/datos.dat — ruta relativa desde la posición actual, al directorio examples y al fichero.
- ../examples/datos.dat — ruta relativa al estilo UNIX, desde la posición actual, al directorio padre (los dos puntos), luego a examples y ahí está el fichero.

Desde un programa en Java puedes pedir al sistema que te diga la ruta absoluta para llegar a algunas carpetas importantes:

- **String System.getProperty("user.dir")** — ruta absoluta a la carpeta actual del usuario.
- **String System.getProperty("user.home")** — ruta absoluta a la carpeta personal del usuario actual.

Para salvar las diferencias entre sistemas, Java tiene la clase

**java.io.File**. Los objetos representan un fichero. Los directorios son tratados también como ficheros.

Un objeto **File** se puede crear con dos constructores:

- **new File(String nombre)** donde el nombre puede ser un nombre, un pathname absoluto o un pathname relativo.
- **new File(File ruta, String nombre)** El primer parámetro es un fichero de tipo directorio que indica la ruta. El segundo es el nombre del fichero.

## CREAR Y ELIMINAR FICHEROS Y DIRECTORIOS.

Podemos crear un fichero en dos pasos. Primero se crea el objeto que encapsula el fichero, por ejemplo, imagina que vamos a crear un fichero llamado miFichero.txt en la carpeta "C:/U07", haríamos:

```
File f = new File("c:\\U07\\miFichero.txt");
```

En segundo lugar, a partir del objeto **File** creamos el fichero físicamente, con la siguiente instrucción, que devuelve un boolean con valor true si se creó correctamente, o false si no se pudo crear:

```
f.createNewFile();
```

Para borrar un fichero, podemos usar la clase **File**, comprobando previamente si existe, del siguiente modo:

```
File f = new File("C:/U07", "agenda.txt");
if ( f.exists() )
  f.delete();
else {
   // Acciones ...
}
```

Para crear un directorio desde Java se puede usar el método **File.mkdir()** y **File.mkdirs()** si el pathname del fichero incluye varias

carpetas que también se deben crear enc aso de no existir.

**EJEMPLO 17**: Para crear directorios, podríamos hacer:

```
public class Ejemplo17 {
   public static void main(String[] args) {
      try {
         String d = "C:/U07";
         String varios = "carpeta1/carpeta2/carpeta3";
         boolean ok = ( new File(d) ).mkdir();
         if( ok )
            System.out.println("Directorio: " + d + " creado");
         // Crear todos los directorios de una ruta
         ok = (new File(varios)).mkdirs();
         if( ok )
          System.out.println("Directorios " + varios +" creados");
      }
      catch(Exception e){
         System.err.println("Error: " + e.getMessage() );
      }
   }
}
```

Para borrar un directorio con Java tenemos que borrar antes cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio con:

```
File[] ficheros = directorio.listFiles();
```

y entonces poder ir borrando. Si uno de los elementos es un directorio, lo podemos averiguar mediante el método isDirectory().

Suponiendo que f sea un objeto File, algunos de sus métodos:

- **f.exists()** — true si el fichero existe en el sistema de ficheros.
- **f.getAbsoluteFile()** y **f.getAbsolutePath()** - devuelve el fichero y el String respectivamente con la ruta absoluta al fichero.

- **f.getCanonicalFile()** y **f.getCanonicalPath()** - devuelve el fichero y el String respectivamente con la ruta canónica (si la ruta da vueltas por las carpetas elimina esas vueltas) al fichero.
- **f.getName()** — String con el nombre.
- **f.getPath()** — String con la ruta.
- **f.lastModified()** — Date de la última modificación.
- **f.length()** — long con la cantidad de bytes del fichero.
- **f.isDirectory()** — true si el fichero es un directorio o false si no es un directorio o no existe.
- **f.isFile()** — true si el fichero no es un directorio o false si lo es.
- **f.isHide()** — true si el fichero está oculto.
- **f.isAbsolute()** — true si la ruta es absoluta.
- **f.getParent()** y **f.getParentFile()** devuelve el String o el File con la ruta de la carpeta padre de f (si existe) o null en otro caso.
- **f.renameTo(File destino)**: cambia nombre y/o ruta del fichero actual. Operaciones dependientes de la plataforma podrían hacer fallar la operación. Devuelve true si ha tenido éxito.
- **f.delete()** — borra el fichero si existe. Devuelve true si la operación tiene éxito.
- **f.getTotalSpace()** y **f.getFreeSpace()** y **f.getUsableSpace()** devuelven un long con la cantidad de bytes totales que tiene la partición de la ruta o 0L si no existe la partición, con los bytes libres que tiene y con los bytes que puede consumir el usuario del programa respectivamente.
- **f.list()** y **f.list(Filename filtro)**— si el fichero es un directorio devuelve un array de Strings (String[]) con los nombres de los ficheros que contiene. Devuelve null en otro caso. La segunda versión filtra los ficheros con el filtro de nombres indicado.
- **f.listFiles()** y **f.listFiles(Filename filtro)** similar, pero devuelve un array de File.

- **f.deleteOnExit()** se borra cuando finaliza la ejecución de la máquina virtual Java. Se usa para ficheros temporales o de maniobra usados por el programa.
- **File.CreateTempFile()** Crea un nuevo fichero con un nombre único, un fichero temporal y devuelve un objeto File que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- **f.setLastModified(long d)** Establece la fecha y la hora de modificación del archivo. Ejemplo:

```
f.setLastModified( new Date().getTime() );
```

- **f.setHide(boolean b)**, **f.setReadable(boolean b)**, **f.setExecutable(boolean b)** y **f.setReadOnly()** y **f.setWritable(boolean b)** establece permisos para el propietario. Hay versiones sobrecargadas con un segundo parámetro booleano indicando si se quiere aplicar a todos los usuarios.
- **f.canExecute()** y **f.canRead()** y **f.canWrita()** consulta las operaciones que pueden realizarse sobre el fichero f.
- **f.mkdir()** Crea un directorio.
- **f.mkdirs()** crea un directorio y los directorios superiores de la ruta si no existen.
- **File.listRoots()** Lista los nombres de archivo de la raíz del sistema de archivos, es un método estático.
- **Path f.toPath()**, **URI f.toURI()** y **URL f.toURL()** - conversión de File a otros mecanismos de acceso a la información.

**EJEMPLO 18**: Programa que liste todos los ficheros del directorio que indique el usuario.

```
/**
* Lista los ficheros de un directorio que indica el usuario
* Si el nombre indicado por el usuario no es un directorio,
* se imprime un mensaje y acaba el programa */
```

```
public class Ejemplo18 {
   public static void main(String[] args) {
      String nombre;  // Nombre del directorio
      File d;         // Directorio
      String[] f;     // Array de nombres de ficheros
      Scanner s;      // Leer del usuario
      s = new Scanner(System.in); // scanner lee entrada estándar
      System.out.print("Teclee un nombre de directorio: ");
      nombre = s.nextLine().trim();
      d = new File(nombre);
      if ( !d.isDirectory() ) {
         if( !d.exists() )
            System.out.println("No existe!");
         else
            System.out.println("No es un directorio.");
      }
      else {
         f = d.list();
         System.out.println("Ficheros en \"" + d + "\":");
         for(int i = 0; i < f.length; i++)
            System.out.println(" " + f[i]);
         }
   } // main()
} // clase
```

## FILTRAR ARCHIVOS

Para buscar sólo aquellos ficheros que tengan determinada característica, por ejemplo, que su extensión sea: .txt, o que se hayan modificado después de una fecha, o los que tienen un tamaño mayor del que indiquemos, etc. Podemos obtener una lista solamente con aquellos que cumplan la condición.

La interface **FilenameFilter** se puede usar para crear filtros que establezcan criterios de filtrado que puedan obtenerse usando el nombre de los ficheros o de la carpeta donde residen. Una clase que lo implemente debe definir e implementar el método:

```
boolean accept(File dir, String nombre);
```

Este método devolverá verdadero (true), en el caso de que el fichero almacenado bajo la carpeta `dir` y cuyo nombre se indica en el parámetro `nombre`, aparezca en la lista de los ficheros del directorio indicado por el parámetro dir.

Aunque la idea es que se base en el nombre, su carpeta y el nombre te permiten localizarlo, así que podrías aprovechar para pedirle otros atributos para decidir si lo aceptas o lo filtras.

La interface **java.io.FileFilter** define el método **boolean accept(File f)** también puede filtrar ficheros.

> *Cuidado!!* hay una clase abstracta que se llama igual en ***javax.swing.filechooser.FileFilter*** que define dos métodos abstractos: ***boolean accept(File f);*** y ***String getDescription();*** se puede utilizar ***FileFilter()*** para instanciarla y la usa FileChooser para filtrar los ficheros que muestra. Ten cuidado porque puedes tener ambiguedades y confusiones en tu código si importas ambas al mismo tiempo.

**EJEMPLO 19**: listar los ficheros de la carpeta c:/U07/ y luego solo los que tengan la extensión ".odt". Usamos try-catch para capturar las posibles excepciones, como que no exista dicha carpeta.

```
import java.io.File;
import java.io.FilenameFilter;

public class Filtro implements FilenameFilter {
   String extension;

   public Filtro(String extension){ this.extension= extension; }

   public boolean accept(File dir, String name){
      return name.endsWith( extension );
   }
```

```
    public static void main(String[] args) {
       try {
          File f= new File("c:\\U07\\.");
          String[] fL1 = f.list();
          String[] fL2 = f.list( new Filtro(".odt") );
          int num = FL1.length;
          if(num < 1)
             System.out.println("No hay archivos que listar");
          else {
             for(int c= 0; c < fL1.length; c++)
                System.out.println( fL1[c] );
             System.out.println( "Aplicando el filtro .odt" );
             for(int c= 0; c < fL2.length; c++)
                System.out.println( fL1[c] );
          }
       }
       catch (Exception ex) {
          System.out.println("Error al buscar en la ruta");
       }
    }
}
```

## 7.2.3 CAJAS DE DIÁLOGO DE FICHEROS.

En las aplicaciones GUI, el usuario cuando deba elegir un fichero con el trabajar espera poder escogerlo de forma gráfica (navegando por las carpetas del sistema de ficheros) sin escribir su nombre o su ruta. No solo es más cómodo. También es un método menos propenso a errores. Swing incluye cajas de diálogo en forma de una clase llamada **JFileChooser** que es parte del paquete **javax.swing**. Los constructores pueden indicar el directorio de inicio o usar el actual si no se indica:

```
new JFileChooser()
new JFileChooser( File directorioInicio )
new JFileChooser( String rutaAlDirectorio )
```

Construir el objeto no hace que aparezca en la pantalla. Hay que pedírselo al objeto. Pero hay dos tipos de diálogos:

- **showOpenDialog()**: permite al usuario abrir un fichero y escoger

uno que exista.

- **showSaveDialog()**: para guardar un fichero que puede o no existir todavía y que se abre para escribir en él.

Un diálogo de ficheros siempre debe tener un padre que es otro elemento gráfico de la aplicación asociada a él. El padre que se suele utilizar es "**this**" o **null** y se pasa como un argumento en la llamada a los métodos.

Como un diálogo es una ventana modal (el resto de elementos se bloquean hasta que no cierres el diálogo), tan llamar al método `showOpenDialog()` como llamar a `showSaveDialog()` devuelven un valor que es una de las siguientes constantes:

- **JFileChooser.CANCEL_OPTION**: el usuario ha pulsado el botón cancelar y no ha elegido un fichero.
- **JFileChooser.ERROR_OPTION**: se ha generado un error.
- **JFileChooser.APPROVE_OPTION**: si el usuario ha seleccionado un fichero. En este caso puedes saber el fichero seleccionando llamando al método **getSelectedFile()** que devuelve un objeto File asociado al fichero.

**EJEMPLO 20**: Hacer un método que lea los datos de un fichero seleccionado de forma gráfica con un diálogo JFileChooser:

```
public void leerFichero() {
   if (fd == null)   // (fd es una var.de instancia) JFileChooser
      fd = new JFileChooser();
   fd.setDialogTitle("Selecciona el fichero a leer");
   fd.setSelectedFile(null);     // No hay ninguno seleccionado
   int opcion = fd.showOpenDialog(this);
   // Al usar "this" el objeto que contiene leerFichero()
   // será de un objeto gráfico, si no es así, usar null
   if (opcion != JFileChooser.APPROVE_OPTION)
      return;    // El usuario cancela
   File f = fd.getSelectedFile();
```

```
        StringReader in;   // (o cualquier otra clase wrapper)
        try {
           FileReader stream = new FileReader(f); // o FileInputStream
           in = new StringReader( stream );
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(null,
                                      "Error abriendo fichero:\n" + e);
           return;
        }
        try {
           // Leer y procesar datos del stream de entrada,...
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(this,
                                     "Error leyendo los datos:\n" + e);
        }
        finally {
           in.close();
        }
    }
```

Como fd es una variable de instancia (no una variable local), le permitirá seguir existiendo aun cuando el método acabe.

**EJEMPLO 21**: Escoger un fichero en el que escribir es similar, pero es buena idea comprobar si el fichero escogido ya existe y preguntar al usuario si realmente quiere reemplazarlo.

```
    public void writeFile() {
       if (fd == null)
          fd = new JFileChooser();
       File f = new File("output.txt"); // nombre por defecto
       fd.setSelectedFile(f);           // fichero por defecto
       fd.setDialogTitle("Seleccione un fichero para escribir");
       int opcion = fd.showSaveDialog(this);
       if (opcion != JFileChooser.APPROVE_OPTION)
          return; // Usuario cancela
       f = fd.getSelectedFile();
       if( f.exists() ) {  // Preguntar si sobreescribe
          int respuesta = JOptionPane.showConfirmDialog(this,
                          "El fichero \"" + f.getName() +
```

```
                        "\" ya existe.\nLo reemplaza?",
                        "Confirm Save",
                        JOptionPane.YES_NO_OPTION,
                        JOptionPane.WARNING_MESSAGE );
        if (respuesta != JOptionPane.YES_OPTION)
            return;     // No quiere
    }
    PrintWriter out; // (o otra clase wrapper)
    try {
        out = new PrintWriter( f );
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog(this,
                                "Error al abrir fichero:\n" + e);
        return;
    }
    try {
        // Escribir datos en el fujo de salida, out
        out.flush();
        out.close();
        if (out.checkError())  // Comprobar
            throw new IOException("Error al escribir fichero.");
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog(this,
              "Error al escribir en fichero:\n" + e);
    }
}
```

Si al navegar no quieres que te muestre todos los ficheros puedes añadirle filtros con el método `addChoosableFileFilter(FileFilter f)`. Un `javax.swing.filechooser.FileFilter` es una clase abstracta que te obliga a implementar:

- `boolean accept(File f)`: devuelve true si el fichero f cumple la condición. Se suele utilizar como condición una o varias extensiones de ficheros (que nos indica el tipo de información que contiene).
- `String getDescription()`: devuelve una descripción del tipo de fichero que selecciona el filtro.

## EJEMPLO 22: Mostrar solamente carpetas y ficheros de tipo .pdf:

```
JFileChooser fc = new JfileChooser();
fc.addChoosableFileFilter(
  new FileFilter() {
     public String getDescription(){ return "Documentos PDF(*.pdf)";}
     public boolean accept(File f) {
         return f.isDirectory() ||
                f.getName().toLowerCase().endsWith(".pdf");
     }
  }
}
);
```

Si por ejemplo queremos dar la opción de escoger cualquier tipo de documento (.pdf, .xls, .docx, .odt, etc.) Podemos hacer un solo filtro con todas las posibilidades o crear un filtro para cada tipo de fichero o hacernos una clase que extienda a la abstracta y que implemente los dos métodos de la abstracta para agilizar su utilización. Bueno, a partir de Java 6 esta clase ya viene de fábrica y se llama `FileNameExtensionFilter` y permite crear filtros de una manera más sencilla indicando la descripción y uno o varios Strings con las extensiones. Por ejemplo:

```
new FileNameExtensionFilter("Documentos PDF", "pdf")
new FileNameExtensionFilter("Imágenes", "jpg", "png", "gif")
```

## EJEMPLO 23: Mostrar carpetas y ficheros de tipo documentos.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import javax.swing.filechooser.FileNameExtensionFilter;
```

```
public class Ejemplo23 extends JFrame {
    private JButton bMostrar;

    public Ejemplo23() {
        super("Demo de Filtros por Tipo");
        setLayout(new FlowLayout());
        bMostrar = new JButton("Mostrar...");
        bMostrar.addActionListener(
          new ActionListener() {
              public void actionPerformed(ActionEvent arg0) {
                  showOpenFileDialog();
              }
          });
        getContentPane().add(bMostrar);
        setSize(300, 100);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                    UIManager.getSystemLookAndFeelClassName());
        } catch(Exception e) { }
        SwingUtilities.invokeLater(
             new Runnable() {
               public void run(){ new Ejemplo23(); }
             });
    }

    private void showOpenFileDialog() {
       JFileChooser fc = new JFileChooser();
       fc.setCurrentDirectory(
                      new File(System.getProperty("user.home")));
       fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
       fc.addChoosableFileFilter(
          new FileNameExtensionFilter("Documentos PDF", "pdf"));
       fc.addChoosableFileFilter(
          new FileNameExtensionFilter("Documentos MS Office",
                                      "docx", "xlsx", "pptx"));
       fc.addChoosableFileFilter(
```

```
                new FileNameExtensionFilter("Imágenes",
                                "jpg", "png", "gif", "bmp"));
            fc.setAcceptAllFileFilterUsed(true);
            int resultado = fc.showOpenDialog(this);
            if(resultado == JFileChooser.APPROVE_OPTION) {
                File fichero = fc.getSelectedFile();
                System.out.println("Seleccionado: " +
                                    fichero.getAbsolutePath());
            }
        }
    }
```

## 7.2.4 FICHEROS COMPRIMIDOS EN .ZIP

Vamos a ver un ejemplo sencillo de cómo crear un fichero zip desde un programa usando Java. Para el ejemplo, tendremos un fichero.txt con un trozo de texto y lo vamos a empaquetar en fichero.zip. Usaremos las clases que nos proporciona Java en su paquete **java.util.zip**.

En primer lugar creamos un **OutputStream** para nuestro fichero zip, de forma que podamos ir poniendo en él nuestros ficheros comprimidos. Este OutputStream se obtiene instanciando un **ZipOutputStream**.

```
ZipOutputStream zos = new ZipOutputStream(
                            new FileOutputStream("fichero.zip")
                      );
```

Podemos indicar el nivel y tipo de compresión que queremos con sus métodos **setMethod()** y **setLevel()**. Los dejaremos en sus valores por defecto que son **DEFLATED** para **setMethod()** (es el algoritmo por defecto) y **setLevel()** con **DEFAULT_COMPRESSION**, ambas constantes definidas en la clase **Deflater**.

```
// Estas son las opciones por defecto, no es
// necesario ponerlas en codigo.
zos.setLevel(Deflater.DEFAULT_COMPRESSION);
zos.setMethod(Deflater.DEFLATED);
```

Una vez hecho esto, sólo debemos ir añadiendo ficheros. Para ello, los

pasos que hay que dar son:

- Indicarle al ZipOutputStream que comenzamos un nuevo fichero (nueva entrada en el fichero comprimido).
- Pasarle los bytes de esa nueva entrada (del fichero). Se pasan sin comprimir y ya se encargará de ir comprimiéndolos según los metemos.
- Indicar a ZipOutputStream que terminamos la nueva entrada.
- Repetir los pasos anteriores mientras queramos seguir metiendo ficheros.

Para indicar a ZipOutputStream que comenzamos a meter un nuevo fichero, ponemos:

```
ZipEntry entrada = new ZipEntry("fichero.txt");
zos.putNextEntry(entrada);
```

El nombre que usemos en ZipEntry es el nombre que queramos que tenga el fichero dentro del zip. Aunque es lo habitual, no tiene por qué coincidir con el nombre del fichero fuera del zip. Si queremos que el fichero comprimido esté dentro de un directorio en el zip, bastará con poner también el path donde lo queremos

```
ZipEntry entrada = new ZipEntry("directorio/fichero.txt");
zos.putNextEntry(entrada);
```

Ahora hay que ir leyendo el fichero normal y pasándole los bytes al ZipOutputStream:

```
FileInputStream fis = new FileInputStream("fichero.txt");
byte[] buffer = new byte[1024];
int leido= 0;
while (0 < (leido=fis.read(buffer))){
   zos.write(buffer,0,leido);
}
```

Símplemente abrimos el fichero normal con un FileInputStream, creamos

un buffer de lectura de 1024 bytes (o el tamaño que consideremos adecuado) y nos metemos en un bucle para ir leyendo y escribiendo en el ZipOutputStream. Una vez que hemos terminado de leer y meter bytes de este fichero, cerramos tanto el fichero como la entrada del zip, indicando así a ZipOutputStream que hemos terminado con este fichero.

```
fis.close();
zos.closeEntry();
```

Repetimos el proceso con todos los ficheros que queramos seguir añadiendo, creando para cada uno de ellos un nuevo ZipEntry, escribiendo los bytes del fichero y cerrando el ZipEntry.

Una vez que acabemos con todos los ficheros, símplemente cerramos el ZipOutputStream:

```
zos.close();
```

## LEER Y EXTRAER UN FICHERO ZIP

Si ya tenemos el fichero zip y lo que queremos es leer su contenido y extraerlo, usaremos la clase **ZipInputStream** pasándole el fichero zip:

```
ZipInputStream zis = new ZipInputStream(
                            new FileInputStream("fichero.zip")
                     );
```

Esta clase tiene métodos para consultar cuántas entradas tiene, cuales son, etc. En este ejemplo que pretende ser sencillo, vamos símplemente a ir recorriendo y extrayendo todas las entradas. Para ir recorriendo las entradas:

```
ZipEntry entrada;
while (null != (entrada=zis.getNextEntry()) ){
   System.out.println( entrada.getName() );
   // Otras tareas...
}
```

A cada ZipEntry podremos interrogarle para saber su tamaño, formato de compresión, etc, etc. Basta ver la API de ZipEntry para ver las posibilidades. Aprovechamos el mismo bucle para ir extrayendo los ficheros:

```
ZipEntry entrada;
while (null != (entrada= zis.getNextEntry()) ){
   System.out.println( entrada.getName() );
   FileOutputStream fos = new FileOutputStream(entrada.getName());
   int leido;
   byte[] buffer = new byte[1024];
   while( 0 < (leido= zis.read(buffer))){
      fos.write(buffer,0,leido);
   }
   fos.close();
   zis.closeEntry();
}
```

Para cada entrada, creamos un FileOutputStream donde escribiremos el fichero descomprimido. Aquí hemos puesto alegremente new FileOutputStream(entrada.getName()), pero en realidad debemos analizar previamente el entrada.getName() para ver si es el nombre de un fichero válido que podamos escribir. Por ejemplo, si como vimos al escribir el zip, el nombre de la entrada fuera "directorio/fichero.txt", antes de abrir deberíamos crear el directorio u obtendremos un error.

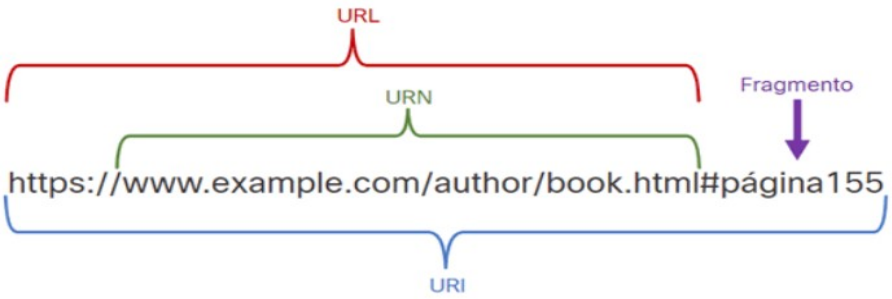
Luego, simplemente se declara un buffer de lectura y nos metemos en un bucle hasta fin de fichero, leyendo del ZipInputStream y escribiendo en el FileOutputStream. Terminado el trasiego, cerramos el FileOutputStream y le indicamos a ZipInputStream que hemos terminado con la entrada, llamando a closeEntry().

Y una vez que salgamos del bucle de entradas (no queden más entradas en el zip), cerramos el ZipOutputStream con **zip.close();**

## 7.2.5. ACCESO A FICHEROS CON URL y URI.

Un **URI** (***uniform resource identifier***) sirve para acceder a un **recurso físico o abstracto de Internet**. Dependiendo de la situación, el recurso puede ser de muchos tipos: puede identificar tanto una página web como al remitente o al destinatario de un mail. Las aplicaciones utilizan URIs para interactuar con recursos o consultar información del mismo.

Un **URL** (***uniform resource locator***) indica dónde se encuentra un recurso. Un **URN** o (**uniform resource name**) es independiente de la ubicación y designa un recurso de forma permanente. Una URL puede identificar un dominio web y un URN puede ser un ISBN. Por tanto URL y URN son un tipo concreto de URI. Del mismo modo, ni URL ni URN son URIs (es más general). El siguiente gráfico puede ayudarte a distinguirlos:




Hay URIs absolutos y relativos. Una URI absoluta tiene el formato genérico siguiente:

```
esquema:parte_específica_del_esquema
```

La parte de la derecha depende de la parte izquierda. Por ejemplo el esquema http usa ciertos formatos y el esquema mailto usa otros formatos. Otra forma genérica de representar una URI que representa una URL

sería este, donde el esquema indica el método de acceso al recurso:

**esquema://<autoridad><ruta>?<consulta>#<fragmento>**

El esquema es el protocolo de comunicaciones que se usa para acceder al recurso (como http, ftp, etc.). Las únicas partes obligatorias en una URI son el esquema y la autoridad representa el nombre del servidor o su dirección IP.

Si la autoridad es un nombre de servidor, podría aparecer de la forma usuario@host:puerto. Si una URL identifica un fichero en el el sistema de ficheros local se usa el esquema file que puede presentar formatos como:

```
file://c:/documents/java.doc.
```

La sintaxis URI utiliza una sintaxis jerárquica dentro de su parte ruta. Cada parte de una ruta se separa por una barra (/).

La parte consulta indica que el recurso se obtiene ejecutando la consulta indicada. Consiste en parejas nombre+valor separadas por un carácter amspersand (&). El nombre se separa del valor por un carácter igual (=). Por ejemplo: id=123&num=5 es una consulta en la que hay dos nombres llamados id y num, el valor para id es 123 y el valor para num es 5.

La parte fragmento identifica un recurso secundario, normalmente una parte del recurso primario que se expresa como otra parte de la URI.

**EJEMPLO 24**: analizar las partes de la URI

http://www.muebles.com/sillas/a.html?id=123#abc

esquema (Scheme): http
Autoridad (Authority): www.muebles.com

| | |
|---|---|
| ruta (Path): | /sillas/a.html |
| consulta (Query): | id=123 |
| Fragmento (Fragment): | abc |

Para usar el carácter espacio en blanco en una URI, se utiliza %20, que es la forma escapada del código en hexadecimal del carácter espacio (valor 32 en ASCII).

Para usar el carácter porcentaje (que sirve de escape) podemos usar %25. Por ejemplo para pasar en una consulta la cadena "5.2%" podemos escribir `http://www.muebles.com/detalle?ratio=5.2%25`

Java representa a URIs y URLs como objetos y aporta las siguientes clases:

- java.net.URI
- java.net.URL
- java.net.URLEncoder
- java.net.URLDecoder

**EJEMPLO 25**: El siguiente trozo de código crea un objeto URI absoluto que utiliza como base, crea un URI relativo y lo resuelve con la base. Muestra detalles de cada uno:

```
import java.net.URI;

public class Ejemplo25 {
  public static void main(String[] args) throws Exception {
    String baseURIStr = "https://aules.edu.gva.es/semipresencial/";
    String relativaURIStr = "mod/forum/discuss.php?d=183770#p385390";
    URI baseURI = new URI(baseURIStr);
    URI relativaURI = new URI(relativaURIStr);
    URI resueltaURI = baseURI.resolve(relativaURI);
    printURIDetalles(baseURI);
    printURIDetalles(relativaURI);
    printURIDetalles(resueltaURI);
  }

  public static void printURIDetalles(URI uri) {
```

```
    System.out.println("  URI: " + uri);
    System.out.println("  Normalizada: " + uri.normalize());
    String partes = "  [Esquema: " + uri.getScheme() + ", Autoridad: "
        + uri.getAuthority() + ", Ruta: " + uri.getPath() +
        ", Consulta:" + uri.getQuery() + ", Fragmento: " +
        uri.getFragment() + "]";
    System.out.println(partes);
  }
}
```

Podemos obtener un objeto `URL` a partir de un objeto `URI` usando su método `toURL()`:

```
URL  baseURL = baseURI.toURL();
```

**EJEMPLO 26:** Ahora vamos a usar el objeto URL.

```
import java.net.URL;

public class Ejemplo26 {
  public static void main(String[] args) throws Exception {
    String baseURLStr = "http://www.ietf.org/rfc/rfc3986.txt";
    String relativaURLStr = "rfc2732.txt";
    URL baseURL = new URL(baseURLStr);
    URL resueltaURL = new URL(baseURL, relativaURLStr);
    System.out.println("URL base: " + baseURL);
    System.out.println("URL relativa: " + relativaURLStr);
    System.out.println("URL resuelta: " + resueltaURL);
  }
}
```

Las clases `URLEncoder` y `URLDecoder` se usan para codificar y decodificar Strings respectivamente.

```
import java.net.URLDecoder;
import java.net.URLEncoder;

public class TestEncoderDecoder {
  public static void main(String[] args) throws Exception {
    String source = "index&^%*a test para 2.5% y &";
    String encoded = URLEncoder.encode(source, "utf-8");
    String decoded = URLDecoder.decode(encoded, "utf-8");
```

```
            System.out.println("Source: " + source);
            System.out.println("Encoded: " + encoded);
            System.out.println("Decoded: " + decoded);
        }
    }
```

## EJEMPLO 27: Accediendo al contenido de una URL

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;

public class Ejemplo27 {
 public static String getContenidoURL(String urlStr) throws Exception{
    BufferedReader br = null;
    URL url = new URL(urlStr);
    InputStream is = url.openStream();
    br = new BufferedReader( new InputStreamReader(is) );
    StringBuilder sb = new StringBuilder();
    String msg = null;
    while ( (msg = br.readLine() ) != null) {
       sb.append(msg);
       sb.append("\n"); // Añade salto de línea
    }
    br.close();
    return sb.toString();
  }

  public static void main(String[] args) throws Exception {
    String urlStr = "https://google.es";
    String contenido = getContenidoURL(urlStr);
    System.out.println(contenido);
  }
}
```

## EJEMPLO 28: Un lector/escritor de datos desde una URL a un fichero.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.InputStream;
import java.io.InputStreamReader;
```

```java
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;
import java.util.Map;

public class Ejemplo28 {
  public static String getContenidoURL(String urlStr)throws Exception{
    URL url = new URL(urlStr);
    URLConnection conex = url.openConnection();
    conex.setDoOutput(true);
    conex.connect();
    OutputStream os = conex.getOutputStream();
    BufferedWriter bw = new BufferedWriter(
                                  new OutputStreamWriter(os));
    bw.write("index.htm");
    bw.flush();
    bw.close();
    printRequestHeaders(conex);
    InputStream ins = conex.getInputStream();
    BufferedReader br = new BufferedReader(
                            new InputStreamReader(ins) );
    StringBuffer sb = new StringBuffer();
    String msg = null;
    while ( (msg = br.readLine()) != null) {
      sb.append(msg);
      sb.append("\n"); // salto de línea
    }
    br.close();
    return sb.toString();
  }

  public static void printRequestHeaders(URLConnection uc) {
    Map headers = uc.getHeaderFields();
    System.out.println(headers);
  }

  public static void main(String[] args) throws Exception {
    String urlStr =
          "https://portal.edu.gva.es/iesserraperenxisa/es/inicio-2/";
    String contenido = getContenidoURL(urlStr);
    System.out.println(contenido);
  }
```

```
}
```

**Clase JarURLConnection**

El objeto `JarURLConnection` nos permite obtener datos que están almacenados dentro de un fichero .jar. El siguiente código muestra como obtener un objeto `JarURLConnection`.

```
String str = "jar:http://sitio.com/fichero.jar!/mi/Abc.class";
URL url = new URL(str);
JarURLConnection  con = (JarURLConnection)url.openConnection();
```

## 7.3 MANIPULAR XML.

XML es un lenguaje textual diseñado para almacenar y transportar datos en texto plano. Sus siglas significan e**X**tensible **M**arkup **L**anguage. Algunas características de XML:

- XML es un lenguaje de marcas.
- XML se basa en tags como HTML.
- Los tags de XML no están predefinidos como en HTML.
- Puedes definir tus propios tags (por eso es extensible).
- Los tags de XML están pensados para ser auto descriptivos.
- XML es el formato recomendado por la W3C para el almacenamiento e intercambio de datos entre procesos.

Ejemplo de fichero XML:

```
<?xml version = "1.0"?>
<Class>
   <Name>Primero</Name>
   <Sections>
      <Section>
         <Name>A</Name>
         <Students>
            <Student>Luis</Student>
            <Student>Maria</Student>
            <Student>Toni</Student>
```

```
            <Student>Santi</Student>
            <Student>Sara</Student>
         </Students>
      </Section>

      <Section>
         <Name>B</Name>
         <Students>
            <Student>Roberto</Student>
            <Student>Julia</Student>
            <Student>Karina</Student>
            <Student>Miguel</Student>
         </Students>
      </Section>
   </Sections>
</Class>
```

**Ventajas:**

- **Independencia de la tecnología**– como es texto plano, puede utilizar cualquier tecnología de almacenamiento y transferencia.
- **Leible por humanos**– entendible por los humanos al usar texto.
- **Extensible**– En XML, puedes personalizar los tags.
- **Permite validación**– Usando XSD, DTD y XML las estructuras se validan de forma sencilla.

**Desventajas:**

- **Sintaxis redundante**– ficheros XML contienen términos muy repetitivos.
- **Parlanchín**– debido a lo repetitivo que es, el peso de los ficheros es alto y aumenta el coste del almacenamiento y transmisión.

El **Parsing** (análisis) de XML se refiere a recorrer un documento XML para acceder, buscar o modificar sus datos. ¿Qué es un Parser XML? Aporta una forma de acceso y modificación de los datos de un documento XML. Java tiene muchas opciones para manipular documentos XML. Algunos parser muy usados:

- **Dom Parser**– carga el documento XML completo y crea un árbol jerárquico en memoria.
- **SAX Parser**– Analiza el documento XML disparando eventos, no carga el documento completo en memoria.
- **JDOM Parser**– como DOM pero más fácil de usar.
- **StAX Parser**– como SAX pero más eficiente.
- **XPath Parser**– se basa en expresiones y se usa mucho junto con XSLT.
- **DOM4J Parser**– Una librería de Java para manipular XML con Xpath y XSLT usando el Framework de colecciones de Java. Ofrece soporte para DOM, SAX y JAXP.

### 7.3.1. EL PARSER DOM.

El **Document Object Model** (DOM) es una recomendación oficial de la W3C. Define una interfaz que permite a los programas acceder y modificar el estilo, la estructura y los contenidos de documentos XML. Los parsers que soportan DOM implementan esta interfaz. ¿Cuándo usarlo?

- Necesitas conocer la estructura de un documento.
- Necesitas mover de sitio partes del documento XML (ordenar ciertos elementos por ejemplo).
- Necesitas usar la información del documento más de una vez.

¿Qué obtienes?
Obtienes una estructura en árbol con todos los elementos del documento. El DOM tiene muchas subrutinas para examinar el contenido y la estructura del documento.

**Ventajas:**
Es una interfaz estándar para manipular la estructuras de los

documentos. Podrías usar otros DOM y obtendrías los mismos resultados.

Interfaces DOM
Las más comunes en Java:

- **Node**– El tipo base de DOM.
- **Element**– la gran mayoría de objetos son Elements.
- **Attr**– Representan el atributo de un elemento.
- **Text**– El contenido actual de un Element o un Attr.
- **Document**– Representa a todo el documento XML. A veces es llamado un árbol DOM.

**Métodos comunes al trabajar con DOM**
Normalmente usarás con mucha frecuencia estos:

- **Document.getDocumentElement()**– devuelve el elemento raíz del documento.
- **Node.getFirstChild()**– devuelve el primer hijo de un nodo.
- **Node.getLastChild()**– devuelve el último hijo de un nodo.
- **Node.getNextSibling()**– devuelve el siguiente hermano de un nodo.
- **Node.getPreviousSibling()**– devuelve el anterior hermano de un nodo.
- **Node.getAttribute(attrName)**– Devuelve el atributo con el nombre indicado.

## PASOS PARA USAR JDOM

Estos son los pasos para parsear documentos con JDOM Parser.

- PASO 1: Importar los packages relacionados con XML.

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
```

- PASO 2: Crear un SAXBuilder.

```
DocumentBuilderFactory factory =
                                DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

- PASO 3: Crear un Document desde un fichero o stream.

```
StringBuilder xml = new StringBuilder();
xml.append("<?xml version="1.0"?> <class> </class>");
ByteArrayInputStream input = new ByteArrayInputStream(
                 xml.toString().getBytes("UTF-8")
                );
Document doc = builder.parse(input);
```

- PASO 4: Extraer el elemento raíz.

```
Element root = document.getDocumentElement();
```

- PASO 5: Examinar los attributos.

```
getAttribute("attributeName");
```

- PASO 6. Examinar sub-elementos

```
//returns a Map (table) of names/values
getAttributes();
getElementsByTagName("subelementName");
getChildNodes();
```

**EJEMPLO 29**: el fichero input.xml que necesita ser parseado.

```
<?xml version = "1.0"?>
<class>
   <student rollno = "393">
      <firstname>dinkar</firstname>
      <lastname>kad</lastname>
      <nickname>dinkar</nickname>
      <marks>85</marks>
   </student>

   <student rollno = "493">
      <firstname>Vaneet</firstname>
      <lastname>Gupta</lastname>
```

```
      <nickname>vinni</nickname>
      <marks>95</marks>
   </student>

   <student rollno = "593">
      <firstname>jasvir</firstname>
      <lastname>singn</lastname>
      <nickname>jazz</nickname>
      <marks>90</marks>
   </student>
</class>
```

```
// Programa: DomParserDemo.java
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;

public class DomParserDemo {

   public static void main(String[] args) {

      try {
         File iF = new File("input.txt");
         DocumentBuilderFactory dbF= DocumentBuilderFactory.newInstance();
         DocumentBuilder dB= dbF.newDocumentBuilder();
         Document doc = dB.parse(inputFile);
         doc.getDocumentElement().normalize();
         System.out.println("Root:" +
                              doc.getDocumentElement().getNodeName());
         NodeList nList = doc.getElementsByTagName("student");
         System.out.println("----------------------------");
         for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            System.out.println("\nActual:" + nNode.getNodeName());
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
               Element eElement = (Element) nNode;
               System.out.println("Student roll no : "
                  + eElement.getAttribute("rollno"));
               System.out.println("First Name : "
                  + eElement
                  .getElementsByTagName("firstname")
                  .item(0)
                  .getTextContent());
               System.out.println("Last Name : "
```

```
                    + eElement
                    .getElementsByTagName("lastname")
                    .item(0)
                    .getTextContent());
                System.out.println("Nick Name : " + eElement
                    .getElementsByTagName("nickname")
                    .item(0)
                    .getTextContent());
                System.out.println("Marks : "
                    + eElement
                    .getElementsByTagName("marks")
                    .item(0)
                    .getTextContent());
             }
          }
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```

Produce el siguiente resultado:

```
Root:class
----------------------------
Actual:student
Student roll no: 393
First Name: dinkar
Last Name: kad
Nick Name: dinkar
Marks: 85

Actual:student
Student roll no: 493
First Name: Vaneet
Last Name: Gupta
Nick Name: vinni
Marks : 95

Actual:student
Student roll no: 593
First Name: jasvir
Last Name: singn
Nick Name: jazz
Marks: 90
```

## CONSULTAR UN DOCUMENTO XML.

### El documento

```
<?xml version = "1.0"?>
<cars>
   <supercars company = "Ferrari">
      <carname type = "formula one">Ferarri 101</carname>
      <carname type = "sports car">Ferarri 201</carname>
      <carname type = "sports car">Ferarri 301</carname>
   </supercars>

   <supercars company = "Lamborgini">
      <carname>Lamborgini 001</carname>
      <carname>Lamborgini 002</carname>
      <carname>Lamborgini 003</carname>
   </supercars>

   <luxurycars company = "Benteley">
      <carname>Benteley 1</carname>
      <carname>Benteley 2</carname>
      <carname>Benteley 3</carname>
   </luxurycars>
</cars>
```

```
// QueryXmlFileDemo.java
package xml;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import java.io.File;

public class QueryXmlFileDemo {

   public static void main(String argv[]) {
      try {
         File inputFile = new File("input.txt");
         DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
         DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
         Document doc = dBuilder.parse(inputFile);
         doc.getDocumentElement().normalize();
         System.out.print("Root element: ");
         System.out.println(doc.getDocumentElement().getNodeName());
```

```
            NodeList nList = doc.getElementsByTagName("supercars");
            System.out.println("----------------------------");
            for (int temp = 0; temp < nList.getLength(); temp++) {
               Node nNode = nList.item(temp);
               System.out.println("\nCurrent Element :");
               System.out.print(nNode.getNodeName());
               if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                  Element eElement = (Element) nNode;
                  System.out.print("company : ");
                  System.out.println(eElement.getAttribute("company"));
                  NodeList carNameList =
eElement.getElementsByTagName("carname");
                  for(int c = 0; c < carNameList.getLength(); c++) {
                     Node node1 = carNameList.item(count);
                     if (node1.getNodeType() == node1.ELEMENT_NODE) {
                        Element car = (Element) node1;
                        System.out.print("car name : ");
                        System.out.println(car.getTextContent());
                        System.out.print("car type : ");
                        System.out.println(car.getAttribute("type"));
                     }
                  }
               }
            }
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
}
```

Produce este resultado:

```
Root element: cars
----------------------------

Current Element:
supercarscompany : Ferrari
car name : Ferarri 101
car type : formula one
car name : Ferarri 201
car type : sports car
car name : Ferarri 301
car type : sports car

Current Element:
supercarscompany: Lamborgini
car name : Lamborgini 001
car type :
```

```
car name : Lamborgini 002
car type :
car name : Lamborgini 003
car type :
```

## CREAR UN DOCUMENTO XML

Necesitamos crear documentos como este:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<cars>
   <supercars company = "Ferrari">
      <carname type = "formula one">Ferrari 101</carname>
      <carname type = "sports">Ferrari 202</carname>
   </supercars>
</cars>
```

```
// CreateXmlFileDemo.java
package xml;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import java.io.File;

public class CreateXmlFileDemo {

   public static void main(String argv[]) {

      try {
         DocumentBuilderFactory dbF=
         DocumentBuilderFactory.newInstance();
         DocumentBuilder dB = dbFactory.newDocumentBuilder();
         Document doc = dB.newDocument();

         // root element
         Element rootE = doc.createElement("cars");
         doc.appendChild(rootE);

         // supercars element
         Element supercar = doc.createElement("supercars");
         rootE.appendChild(supercar);
```

```
            // setting attribute to element
            Attr attr = doc.createAttribute("company");
            attr.setValue("Ferrari");
            supercar.setAttributeNode(attr);

            // carname element
            Element carname = doc.createElement("carname");
            Attr attrType = doc.createAttribute("type");
            attrType.setValue("formula one");
            carname.setAttributeNode(attrType);
            carname.appendChild(doc.createTextNode("Ferrari 101"));
            supercar.appendChild(carname);

            Element carname1 = doc.createElement("carname");
            Attr attrType1 = doc.createAttribute("type");
            attrType1.setValue("sports");
            carname1.setAttributeNode(attrType1);
            carname1.appendChild(doc.createTextNode("Ferrari 202"));
            supercar.appendChild(carname1);

            // write the content into xml file
            TransformerFactory tF = TransformerFactory.newInstance();
            Transformer t = tF.newTransformer();
            DOMSource source = new DOMSource(doc);
            StreamResult result = new StreamResult(new File("C:\\cars.xml"));
            t.transform(source, result);

            // Output to console for testing
            StreamResult consoleResult = new StreamResult(System.out);
            t.transform(source, consoleResult);
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

Produce el resultado:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<cars>
   <supercars company = "Ferrari">
      <carname type = "formula one">Ferrari 101</carname>
      <carname type = "sports">Ferrari 202</carname>
   </supercars>
</cars>
```

## MODIFICAR UN DOCUMENTO XML

Este es el fichero (input.xml) de entrada a modificar:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<cars>
   <supercars company = "Ferrari">
      <carname type = "formula one">Ferrari 101</carname>
      <carname type = "sports">Ferrari 202</carname>
   </supercars>

   <luxurycars company = "Benteley">
      <carname>Benteley 1</carname>
      <carname>Benteley 2</carname>
      <carname>Benteley 3</carname>
   </luxurycars>
</cars>
```

```
// ModifyXmlFileDemo.java
package xml;

import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.tF;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class ModifyXmlFileDemo {

   public static void main(String argv[]) {
    try {
       File iF = new File("input.xml");
       DocumentBuilderFactory dBF =
                           DocumentBuilderFactory.newInstance();
       DocumentBuilder docB = dBF.newDocumentBuilder();
       Document doc = docB.parse(iF);
       Node cars = doc.getFirstChild();
       Node supercar=
                    doc.getElementsByTagName("supercars").item(0);
       // Modifica el atributo supercar
       NamedNodeMap attr = supercar.getAttributes();
```

```
            Node nodeAttr = attr.getNamedItem("company");
            nodeAttr.setTextContent("Lamborigini");
            // Iterar por los nodos hijo de supercar
            NodeList list = supercar.getChildNodes();
            for(int temp = 0; temp < list.getLength(); temp++) {
               Node node = list.item(temp);
               if(node.getNodeType() == Node.ELEMENT_NODE) {
                  Element eE = (Element) node;
                  if("carname".equals(eE.getNodeName())) {
                     if("Ferrari 101".equals(eE.getTextContent())) {
                          eE.setTextContent("Lamborigini 001");
                     }
                     if("Ferrari 202".equals(eE.getTextContent()))
                        eE.setTextContent("Lamborigini 002");
                     }
                  }
               }
               NodeList childNodes = cars.getChildNodes();
              for(int c = 0; c < childNodes.getLength(); c++) {
                 Node node = childNodes.item(c);
                 if("luxurycars".equals(node.getNodeName()))
                    cars.removeChild(node);
              }
              // Escribir contenido en consola
              tF tF = tF.newInstance();
              Transformer t = tF.newTransformer();
              DOMSource source = new DOMSource(doc);
              System.out.println("--------Fichero Modificado--------");
              StreamResult cR = new StreamResult(System.out);
              t.transform(source, cR);
           }
           catch (Exception e) {
              e.printStackTrace();
           }
        }
    }
```

Genera:

```
--------Fichero Modificado--------
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>

<cars>
   <supercars company = "Lamborigini">
      <carname type = "formula one">Lamborigini 001</carname>
      <carname type = "sports">Lamborigini 002</carname>
   </supercars>
</cars>
```

## 7.3.2 EL PARSER XML SAX.

SAX (**Simple API for XML**) es un parser basado en eventos. Al contrario que DOM, SAX no crea un árbol, es una interfaz de tipo flujo para XML, lo que significa que las aplicaciones que usan SAX reciben notificaciones a medida que van procesando el documento XML al comenzar a procesar un elemento, un atributo, en orden secuencial comenzando por el principio y acabando con el cierre del elemento raíz (ROOT).

- Lee el documetno de principio a fin reconociendo tokens que hacen un XML bien formado.
- Los tokens se procesan en el orden en que aparecen en el documento.
- La aplicación informa de la naturaleza de cada token que el parser encuentra.
- El aporgama aporta manejadores de eventos que registra en el parser.
- Cuando el token es identificado, se ejecutan llamadas a métodos a los que se pasa la información importante sobre ellos.

¿Cuándo es interesante usar este parser? Cuando...

- Procesas el documento XML de forma lineal (de principio a fin).
- El documento no tiene excesivas anidaciones.
- Los XML son muy largos y el DOM consume mucha memoria.
- El trabajo se hace procesando una parte del XML.

Ventajas de SAX:

- Los datos están disponibles tan pronto como el parser los alcanza, así que se adapta bien a los flujos.

Desventajas de SAX:

- No tenemos acceso aleatorio al XML.
- Si necesitas recordar los fatos que ya has parseado o mantener un orden, debes hacerlo tu mismo.

**ContentHandler Interface**

Tiene los métodos que usa SAX para avisar al programa de que encuentra un elemento XML.

- **void startDocument()**– Called at the beginning of a document.
- **void endDocument()**– Called at the end of a document.
- **void startElement(String uri, String localName, String qName, Attributes atts)**– Called at the beginning of an element.
- **void endElement(String uri, String localName,String qName)** – Called at the end of an element.
- **void characters(char[] ch, int start, int length)**– Called when character data is encountered.
- **void ignorableWhitespace( char[] ch, int start, int length)**– cuando hay un DTD y se detectan espacios ignorables.
- **void processingInstruction(String target, String data)**– Called when a processing instruction is recognized.
- **void setDocumentLocator(Locator locator))** – Provides a Locator that can be used to identify positions in the document.
- **void skippedEntity(String name)**– Called when an unresolved entity is encountered.
- **void startPrefixMapping(String prefix, String uri)**– Called when a new namespace mapping is defined.
- **void endPrefixMapping(String prefix)** – Called when a namespace definition ends its scope.

**Attributes Interface**

Métodos para procesar los atributos conectados a un elemento:

- **int getLength()**– Returns number of attributes.
- **String getQName(int index)**
- **String getValue(int index)**
- **String getValue(String qname)**

## PARSEAR CON SAX

Tenemos este fichero que hay que parsear:

```
<?xml version = "1.0"?>
<class>
   <student rollno = "393">
      <firstname>dinkar</firstname>
      <lastname>kad</lastname>
      <nickname>dinkar</nickname>
      <marks>85</marks>
   </student>

   <student rollno = "493">
      <firstname>Vaneet</firstname>
      <lastname>Gupta</lastname>
      <nickname>vinni</nickname>
      <marks>95</marks>
   </student>

   <student rollno = "593">
      <firstname>jasvir</firstname>
      <lastname>singn</lastname>
      <nickname>jazz</nickname>
      <marks>90</marks>
   </student>
</class>
```

```
// UserHandler.java
package xml;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class UserHandler extends DefaultHandler {

   boolean bFirstName = false;
   boolean bLastName = false;
   boolean bNickName = false;
   boolean bMarks = false;
```

```
   @Override
   public void startElement(String uri,
   String localName, String qName, Attributes attributes) throws SAXException {

      if (qName.equalsIgnoreCase("student")) {
         String rollNo = attributes.getValue("rollno");
         System.out.println("Roll No : " + rollNo);
      } else if (qName.equalsIgnoreCase("firstname")) {
         bFirstName = true;
      } else if (qName.equalsIgnoreCase("lastname")) {
         bLastName = true;
      } else if (qName.equalsIgnoreCase("nickname")) {
         bNickName = true;
      }
      else if (qName.equalsIgnoreCase("marks")) {
         bMarks = true;
      }
   }

   @Override
   public void endElement(String uri,
   String localName, String qName) throws SAXException {
      if (qName.equalsIgnoreCase("student")) {
         System.out.println("End Element :" + qName);
      }
   }

   @Override
   public void characters(char ch[], int start, int length) throws SAXException {

      if (bFirstName) {
         System.out.println("First Name: "
            + new String(ch, start, length));
         bFirstName = false;
      } else if (bLastName) {
         System.out.println("Last Name: " + new String(ch,start,length));
         bLastName = false;
      } else if (bNickName) {
         System.out.println("Nick Name: " + new String(ch,start,length));
         bNickName = false;
      } else if (bMarks) {
         System.out.println("Marks: " + new String(ch, start, length));
         bMarks = false;
      }
   }
}

// SAXParserDemo.java
```

```
package xml;

import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserDemo {

   public static void main(String[] args) {

      try {
         File inputFile = new File("input.txt");
         SAXParserFactory factory = SAXParserFactory.newInstance();
         SAXParser saxParser = factory.newSAXParser();
         UserHandler userhandler = new UserHandler();
         saxParser.parse(inputFile, userhandler);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}

class UserHandler extends DefaultHandler {

   boolean bFirstName = false;
   boolean bLastName = false;
   boolean bNickName = false;
   boolean bMarks = false;

   @Override
   public void startElement(
      String uri, String localName, String qName, Attributes attributes)
      throws SAXException {

      if (qName.equalsIgnoreCase("student")) {
         String rollNo = attributes.getValue("rollno");
         System.out.println("Roll No : " + rollNo);
      } else if (qName.equalsIgnoreCase("firstname")) {
         bFirstName = true;
      } else if (qName.equalsIgnoreCase("lastname")) {
         bLastName = true;
      } else if (qName.equalsIgnoreCase("nickname")) {
         bNickName = true;
      }
      else if (qName.equalsIgnoreCase("marks")) {
```

```
            bMarks = true;
         }
      }

      @Override
      public void endElement(String uri,
         String localName, String qName) throws SAXException {

         if (qName.equalsIgnoreCase("student")) {
            System.out.println("End Element :" + qName);
         }
      }

      @Override
      public void characters(char ch[], int start, int length) throws SAXException {

         if (bFirstName) {
           System.out.println("First Name: " + new String(ch,start,length));
            bFirstName = false;
         } else if (bLastName) {
            System.out.println("Last Name: " + new String(ch,start,length));
            bLastName = false;
         } else if (bNickName) {
            System.out.println("Nick Name: " + new String(ch,start,length));
            bNickName = false;
         } else if (bMarks) {
            System.out.println("Marks: " + new String(ch, start, length));
            bMarks = false;
         }
      }
}
```

Genera el resultado:

```
Roll No : 393
First Name: dinkar
Last Name: kad
Nick Name: dinkar
Marks: 85
End Element :student
Roll No : 493
First Name: Vaneet
Last Name: Gupta
Nick Name: vinni
Marks: 95
End Element :student
Roll No : 593
First Name: jasvir
Last Name: singn
```

```
Nick Name: jazz
Marks: 90
End Element :student
```

## CONSULTAR CON SAX

Ejemplo: queremos consultar los datos del rollno: 393 de este fichero.

```
<?xml version = "1.0"?>
<class>
   <student rollno = "393">
      <firstname>dinkar</firstname>
      <lastname>kad</lastname>
      <nickname>dinkar</nickname>
      <marks>85</marks>
   </student>

   <student rollno = "493">
      <firstname>Vaneet</firstname>
      <lastname>Gupta</lastname>
      <nickname>vinni</nickname>
      <marks>95</marks>
   </student>

   <student rollno = "593">
      <firstname>jasvir</firstname>
      <lastname>singn</lastname>
      <nickname>jazz</nickname>
      <marks>90</marks>
   </student>
</class>
```

```
// UserHandler.java
package xml;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class UserHandler extends DefaultHandler {

   boolean bFirstName = false;
   boolean bLastName = false;
   boolean bNickName = false;
   boolean bMarks = false;
   String rollNo = null;

   @Override
```

```
public void startElement(
   String uri, String localName, String qName, Attributes attributes)
   throws SAXException {

   if (qName.equalsIgnoreCase("student")) {
      rollNo = attributes.getValue("rollno");
   }
   if(("393").equals(rollNo) &&
      qName.equalsIgnoreCase("student")) {
      System.out.println("Start Element :" + qName);
   }
   if (qName.equalsIgnoreCase("firstname")) {
      bFirstName = true;
   } else if (qName.equalsIgnoreCase("lastname")) {
      bLastName = true;
   } else if (qName.equalsIgnoreCase("nickname")) {
      bNickName = true;
   }
   else if (qName.equalsIgnoreCase("marks")) {
      bMarks = true;
   }
}

@Override
public void endElement(
   String uri, String localName, String qName) throws SAXException {

   if (qName.equalsIgnoreCase("student")) {
      if(("393").equals(rollNo) && qName.equalsIgnoreCase("student"))
      System.out.println("End Element :" + qName);
   }
}

@Override
public void characters(char ch[], int start, int length)
throws SAXException {
   if (bFirstName && ("393").equals(rollNo)) {
     //age element, set Employee age
     System.out.println("First Name: " + new String(ch,start,length));
     bFirstName = false;
   } else if (bLastName && ("393").equals(rollNo)) {
     System.out.println("Last Name: " + new String(ch,start,length));
      bLastName = false;
   } else if (bNickName && ("393").equals(rollNo)) {
     System.out.println("Nick Name: " + new String(ch, start,length));
     bNickName = false;
   } else if (bMarks && ("393").equals(rollNo)) {
      System.out.println("Marks: " + new String(ch, start, length));
      bMarks = false;
```

```
      }
   }
}
```

```
// SAXQueryDemo.java
package xml;

import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXQueryDemo {

   public static void main(String[] args) {

      try {
         File inputFile = new File("input.txt");
         SAXParserFactory factory =
                              SAXParserFactory.newInstance();
         SAXParser saxParser = factory.newSAXParser();
         UserHandler userhandler = new UserHandler();
         saxParser.parse(inputFile, userhandler);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}

class UserHandler extends DefaultHandler {
   boolean bFirstName = false;
   boolean bLastName = false;
   boolean bNickName = false;
   boolean bMarks = false;
   String rollNo = null;

   @Override
   public void startElement(String uri,
      String localName, String qName, Attributes attributes)
      throws SAXException {

      if (qName.equalsIgnoreCase("student")) {
         rollNo = attributes.getValue("rollno");
      }
      if( ("393").equals(rollNo) &&
                           qName.equalsIgnoreCase("student")) {
```

```
            System.out.println("Start Element :" + qName);
         }
         if (qName.equalsIgnoreCase("firstname")) {
            bFirstName = true;
         } else if (qName.equalsIgnoreCase("lastname")) {
            bLastName = true;
         } else if (qName.equalsIgnoreCase("nickname")) {
            bNickName = true;
         }
         else if (qName.equalsIgnoreCase("marks")) {
            bMarks = true;
         }
      }

      @Override
      public void endElement(String uri, String localName, String qName)
   throws SAXException {
            if (qName.equalsIgnoreCase("student")) {
            if(("393").equals(rollNo)
               && qName.equalsIgnoreCase("student"))
               System.out.println("End Element :" + qName);
         }
      }

      @Override
      public void characters(
         char ch[], int start, int length) throws SAXException {

         if (bFirstName && ("393").equals(rollNo)) {
            //age element, set Employee age
            System.out.println("First Name: " + new String(ch, start,
   length));
            bFirstName = false;
         } else if (bLastName && ("393").equals(rollNo)) {
            System.out.println("Last Name: " + new String(ch, start,
   length));
            bLastName = false;
         } else if (bNickName && ("393").equals(rollNo)) {
            System.out.println("Nick Name: " + new String(ch, start,
   length));
            bNickName = false;
         } else if (bMarks && ("393").equals(rollNo)) {
            System.out.println("Marks: " + new String(ch, start, length));
            bMarks = false;
         }
      }
   }
```

Genera el siguiente resultado:

```
Start Element :student
First Name: dinkar
Last Name: kad
Nick Name: dinkar
Marks: 85
End Element :student
```

***Nota:*** *Es mejor usar el parser StAX para crear documentos XML que el parser SAX.*

## MODIFICAR UN DOCUMENTO CON SAX

Partimos de este fichero XML y debemos añadir **<Result> Pass <Result/>** al final del tag **</marks>**.

```
<?xml version = "1.0"?>
<class>
   <student rollno = "393">
      <firstname>dinkar</firstname>
      <lastname>kad</lastname>
      <nickname>dinkar</nickname>
      <marks>85</marks>
   </student>

   <student rollno = "493">
      <firstname>Vaneet</firstname>
      <lastname>Gupta</lastname>
      <nickname>vinni</nickname>
      <marks>95</marks>
   </student>

   <student rollno = "593">
      <firstname>jasvir</firstname>
      <lastname>singn</lastname>
      <nickname>jazz</nickname>
      <marks>90</marks>
   </student>
</class>
```

```
// Aplicación SAXModifyDemo.java
package xml;

import java.io.*;
```

```
import org.xml.sax.*;
import javax.xml.parsers.*;
import org.xml.sax.helpers.DefaultHandler;

public class SAXModifyDemo extends DefaultHandler {
   static String displayText[] = new String[1000];
   static int nL = 0;    // Nºs de linea
   static String indenta = "";

   public static void main(String args[]) {
      try {
         File inputFile = new File("input.txt");
         SAXParserFactory factory =
         SAXParserFactory.newInstance();
         SAXModifyDemo obj = new SAXModifyDemo();
         obj.childLoop(inputFile);
         FileWriter filewriter = new FileWriter("newfile.xml");
         for(int i = 0; i < nLs; i++) {
            filewriter.write(displayText[i].toCharArray());
            filewriter.write('\n');
            System.out.println(displayText[i].toString());
         }
         filewriter.close();
      }
      catch (Exception e) {
         e.printStackTrace(System.err);
      }
   }

   public void childLoop(File input) {
      DefaultHandler handler = this;
      SAXParserFactory fac = SAXParserFactory.newInstance();
      try {
         SAXParser saxParser = fac.newSAXParser();
         saxParser.parse(input, handler);
      } catch (Throwable t) {}
   }

   public void startDocument() {
      displayText[nL] = indenta;
      displayText[nL] += "<?xml version = \"1.0\" encoding = \""+
         "UTF-8" + "\"?>";
      nL++;
   }

   public void processingInstruction(String target, String data) {
      displayText[nL] = indenta;
      displayText[nL] += "<?";
      displayText[nL] += target;
```

```
        if (data != null && data.length() > 0) {
           displayText[nL] += ' ';
           displayText[nL] += data;
        }
        displayText[nL] += "?>";
        nL++;
    }

    public void startElement(String uri, String localName, String
qualifiedName, Attributes attributes) {
        displayText[nL] = indenta;
        indenta += "    ";
        displayText[nL] += '<';
        displayText[nL] += qualifiedName;
        if (attributes != null) {
           int numberAttributes = attributes.getLength();
           for (int i = 0; i < numberAttributes; i++) {
              displayText[nL] += ' ';
              displayText[nL] += attributes.getQName(i);
              displayText[nL] += "=\"";
              displayText[nL] += attributes.getValue(i);
              displayText[nL] += '"';
           }
        }
        displayText[nL] += '>';
        nL++;
    }

    public void characters(char characters[], int start, int length) {
      String cData = (new String(characters, start,length)).trim();
      if(cData.indexOf("\n") < 0 && cData.length() > 0) {
          displayText[nL] = indenta;
          displayText[nL] += cData;
          nL++;
      }
    }

    public void endElement(String uri, String localName, String
qualifiedName) {
        indenta = indenta.substring(0, indenta.length() - 4) ;
        displayText[nL] = indenta;
        displayText[nL] += "</";
        displayText[nL] += qualifiedName;
        displayText[nL] += '>';
        nL++;

        if (qualifiedName.equals("marks")) {
           startElement("", "Result", "Result", null);
```

```
                characters("Pass".toCharArray(), 0, "Pass".length());
                endElement("", "Result", "Result");
            }
        }
    }
```

Que produce este resultado:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<class>
   <student rollno = "393">
      <firstname>
         dinkar
      </firstname>
      <lastname>
         kad
      </lastname>
      <nickname>
         dinkar
      </nickname>
      <marks>
         85
      </marks>
      <Result>
         Pass
      </Result>
   </student>
   <student rollno = "493">
      <firstname>
         Vaneet
      </firstname>
      <lastname>
         Gupta
      </lastname>
      <nickname>
         vinni
      </nickname>
      <marks>
         95
      </marks>
      <Result>
         Pass
      </Result>
   </student>
   <student rollno = "593">
      <firstname>
         jasvir
      </firstname>
      <lastname>
```

```
            singn
         </lastname>
         <nickname>
            jazz
         </nickname>
         <marks>
            90
         </marks>
         <Result>
            Pass
         </Result>
      </student>
   </class>
```

## 7.3.3. EL PARSER JDOM.

JDOM es un parser open source basado en la librería de Java. Es una API amigable optimizada para usarla en colecciones como List y Arrays. JDOM trabaja con DOM y SAX y combina lo mejor de las dos: consume poca memoria y es casi tan rápida como SAX.

**Configurar el entorno**

Tener jdom.jar en el classpath de la aplicación. Descargable desde jdom-2.0.5.zip.

¿Cuándo es bueno usarlo?

- Necesitas saber algo de la estructura del documento.
- Necesitas mover partes del documento.
- Necesitas acceder a los datos del documento más de una vez.
- Quieres hacer parsing optimizado.

**¿Qué obtienes?**

Puedes obtener un árbol con la estructura y tienes métodos para acceder a la información de cada elemento.

**CLASES DE JDOM**

Las más usadas:

- **Document**– Representa al documento XML(un DOM tree).
- **Element**– Representa un elemento XML.
- **Attribute**– Representa un atributo de un elemento.
- **Text**– Representa el texto de un tag XML.
- **Comment**– Representa los comentarios.

**MÉTODOS JDOM**

Los que se usan con más frecuencia:

- **SAXBuilder.build(xmlSource)()**– crea el JDOM desde XML.
- **Document.getRootElement()**– Obtiene el elemento raíz.
- **Element.getName()**– nombre del noodo XML.
- **Element.getChildren()**– hijos directos de un elemento.
- **Node.getChildren(Name)**– todos los nodos hijo de un nombre.
- **Node.getChild(Name)**– el primer nodo hijo de un nombre.

## 7.3.4. EL PARSER XPATH.

XPath es una recomendación oficial de la W3C. Define un lenguaje para encontrar información en documentos XML.

**Expresiones XPath**

XPath usa expresiones path para seleccionar uno o una lista de nodos de un documento XML.

| Sr.No | Expression & Description |
|---|---|
| 1 | Node-name los nodos que se llamen así "nodename" |
| 2 | / la selección comienza desde la raíz del documento |
| 3 | // la selección comienza desde el nodo actual que case con la selección |
| 4 | . Selecciona el nodo actual |
| 5 | .. Selecciona el nodo padre del actual |
| 6 | @ Selecciona atributos |

Ejemplo de expresiones:

**student** -> Selecciona todos los nodos de nombre "student"

**class/student** -> Selecciona todos los elementos hijos de la clase

**//student** -> Selecciona todos los elementos student que no lo hayan sido ya

**Predicados**

Sirven para escoger nodos que contengan valores concretos. Se definen usando el operador [...].

| Expression | Result |
|---|---|
| /class/student[1] | Selecciona el primer elemento student que sea hijo del elemento class. |
| /class/student[last()] | Selecciona el último elemento student que sea hijo de un elemento class |
| /class/student[last()-1] | penúltimo elemento student que sea hijo de un elemento class |
| //student[@rollno = '493'] | Todos los student que tengan el atributo rollno con valor '493' |

**PASOS PARA USAR XPath**

Estos son los pasos que hay que seguir para utilizar el parser Xpath:

- Importar los paquetes XML.
- Crear un DocumentBuilder.
- Crear un Document desde un fichero o stream.
- Crear un objeto Xpath y una expresión XPath path.
- Compilar la expresión XPath usando Xpath.compile() y obtener la lista de nodos al evaluarla con XPath.evaluate().
- Iteratar sobre la lista de nodos.
- Examinar los atributos de los nodos.

•Examinar los subelementos.

```
// Importar paquetes XML
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import javax.xml.xpath.*;
import java.io.*;
// Crear un DocumentBuilder
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
// Crear un Document desde un fichero o stream
StringBuilder xmlSB = new StringBuilder();
xmlSB.append("<?xml version = "1.0"?> <class> </class>");
ByteArrayInputStream input =  new ByteArrayInputStream(
   xmlSB.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);
// Crear un XPath
XPath xPath =  XPathFactory.newInstance().newXPath();
// Preparar la expresión y evaluarla
String expresion = "/class/student";
NodeList nodeList = (NodeList) xPath.compile(expresion).evaluate(
   doc, XPathConstants.NODESET);
// Iterar sobre la lista de nodos
for (int i = 0; i < nodeList.getLength(); i++) {
   Node nNode = nodeList.item(i);
   ...
}
// Examinar atributos. Por ejemplo coger uno en concreto
getAttribute("attributeName");

// devolver un Map (tabla) de parejas nambre/valor
getAttributes();

// Examinar sub elementos (un determinado nombre)
getElementsByTagName("subelementName");

// Una lista de todos los nodos
getChildNodes();
```

## PARSEAR UN DOCUMENTO CON XPATH

Necesitamos parsear este documento XML:

```
<?xml version = "1.0"?>
<class>
```

```
   <student rollno = "393">
      <firstname>dinkar</firstname>
      <lastname>kad</lastname>
      <nickname>dinkar</nickname>
      <marks>85</marks>
   </student>

   <student rollno = "493">
      <firstname>Vaneet</firstname>
      <lastname>Gupta</lastname>
      <nickname>vinni</nickname>
      <marks>95</marks>
   </student>

   <student rollno = "593">
      <firstname>jasvir</firstname>
      <lastname>singh</lastname>
      <nickname>jazz</nickname>
      <marks>90</marks>
   </student>
</class>
```

```
// XPathParserDemo.java
package xml;

import java.io.File;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.xml.sax.SAXException;

public class XPathParserDemo {

   public static void main(String[] args) {

      try {
         File inputFile = new File("input.txt");
         DocumentBuilderFactory dbFactory =
```

```
DocumentBuilderFactory.newInstance();
         DocumentBuilder dBuilder;

         dBuilder = dbFactory.newDocumentBuilder();

         Document doc = dBuilder.parse(inputFile);
         doc.getDocumentElement().normalize();

         XPath xPath =  XPathFactory.newInstance().newXPath();

         String expression = "/class/student";
         NodeList nodeList = (NodeList)
xPath.compile(expression).evaluate(
            doc, XPathConstants.NODESET);

         for (int i = 0; i < nodeList.getLength(); i++) {
            Node nNode = nodeList.item(i);
            System.out.println("\nCurrent Element :" +
                               nNode.getNodeName());
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
               Element eElement = (Element) nNode;
               System.out.println("Student roll no :" +
                                  eElement.getAttribute("rollno"));
               System.out.println("First Name : "
                  + eElement
                  .getElementsByTagName("firstname")
                  .item(0)
                  .getTextContent());
               System.out.println("Last Name : "
                  + eElement
                  .getElementsByTagName("lastname")
                  .item(0)
                  .getTextContent());
               System.out.println("Nick Name : " + eElement
                  .getElementsByTagName("nickname")
                  .item(0)
                  .getTextContent());
               System.out.println("Marks : " + eElement
                  .getElementsByTagName("marks")
                  .item(0)
                  .getTextContent());
            }
         }
      } catch (ParserConfigurationException e) {
         e.printStackTrace();
      } catch (SAXException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
```

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

Produce este resultado:

```
Current Element :student
Student roll no : 393
First Name : dinkar
Last Name : kad
Nick Name : dinkar
Marks : 85

Current Element :student
Student roll no : 493
First Name : Vaneet
Last Name : Gupta
Nick Name : vinni
Marks : 95

Current Element :student
Student roll no : 593
First Name : jasvir
Last Name : singh
Nick Name : jazz
Marks : 90
```

### 7.3.5. MANIPULAR JSON EN JAVA.

JSON es otro formato que se utiliza para intercambiar información entre programas, bases de datos. Puede representar objetos de forma textual, que al ser serializados pueden almacenarse para hacerse permanentes, o enviarse por la red de un lugar a otro. El formato JSON básicamene consiste en representar con texto la estructura de un objeto:

| | |
|---|---|
| Objeto | { miembro1, miembro2, ... } |
| miembro | "nombre":valor |
| Array | [valor1, valor2, ... ] |
| cadena | " " |

Antes de codificar y decodificar en formato JSON, hay que instalar alguno de los módulos disponibles, por ejemplo JSON.simple y tener añadido el lugar donde se almacena el fichero json-simple-1.1.1.jar a la variable de entorno CLASSPATH.

Mapeo de tipos entre JSON y Java
JSON.simple transforma cada tipo de izquierda a derecha (decodifica) y de derecha a izquierda (codifica).

| JSON | Java |
|---|---|
| string | java.lang.String |
| number | java.lang.Number |
| true\|false | java.lang.Boolean |
| null | null |
| array | java.util.List |
| object | java.util.Map |

Para codificar la clase **java.util.List** se usa **org.json.simple.JSONArray** y para **java.util.Map** se utiliza la clase **org.json.simple.JSONObject**.

## GENERAR JSON DESDE JAVA (CODIFICAR)

Vamos a codificar en un fichero JSON, un objeto Java usando el objeto JSONObject que es una subclase de java.util.HashMap. Si necesitas orden en los elementos, debes usar el método JSONValue.toJSONString(map) que implementa de forma ordenada un map como java.util.LinkedHashMap.

```
import org.json.simple.JSONObject;

class JsonEncodeDemo {

   public static void main(String[] args) {
      JSONObject obj = new JSONObject();
      obj.put("nombre", "Juan");
      obj.put("numero", new Integer(100));
```

```
        obj.put("balance", new Double(1000.21));
        obj.put("es_vip", new Boolean(true));
        System.out.print(obj);
    }
}
```

Genera el siguiente resultado:

```
{"balance": 1000.21, "numero":100, "es_vip":true, "nombre":"Juan"}
```

El mismo ejemplo pero ahora usando un flujo:

```
import org.json.simple.JSONObject;

class JsonEncodeDemo {

   public static void main(String[] args) {
      JSONObject obj = new JSONObject();
      obj.put("nombre","Juan");
      obj.put("numero",new Integer(100));
      obj.put("balance",new Double(1000.21));
      obj.put("es_vip",new Boolean(true));
      StringWriter out = new StringWriter();
      obj.writeJSONString(out);
      String json = out.toString();
      System.out.print(json);
   }
}
```

## GENERAR JAVA A PARTIR DE JSON (DECODIFICAR)

El siguiente ejemplo utiliza **JSONObject** (un java.util.Map) y **JSONArray** (un java.util.List), así que puedes trabajar con el objeto usando las operaciones de las colecciones Map y List.

```
import org.json.simple.JSONObject;
import org.json.simple.JSONArray;
import org.json.simple.parser.ParseException;
import org.json.simple.parser.JSONParser;

class JsonDecodeDemo {

   public static void main(String[] args) {
      JSONParser parser = new JSONParser();
      // El objeto JSON a recuperar
```

```
        String s = "[0, " +
                   "{\"1\":{\"2\":{\"3\":{\"4\":[5,{\"6\":7}]}}}}" +
                   "]";

        try{
           Object obj = parser.parse(s);  // Crear el objeto
           JSONArray array = (JSONArray)obj;
           System.out.println("El segundo elemento del array");
           System.out.println( array.get(1) );
           System.out.println();
           JSONObject obj2 = (JSONObject)array.get(1);
           System.out.println("Campo \"1\"");
           System.out.println( obj2.get("1") );
           s = "{}";
           obj = parser.parse(s);
           System.out.println(obj);
           s = "[5,]";
           obj = parser.parse(s);
           System.out.println(obj);
           s = "[5,,2]";
           obj = parser.parse(s);
           System.out.println(obj);
        }
        catch(ParseException pe) {
           System.out.println("posición: " + pe.getPosition());
           System.out.println(pe);
        }
     }
}
```

El resultado que genera:

```
El segundo elemento del array
{"1":{"2":{"3":{"4":[5,{"6":7}]}}}}

Campo "1"
{"2":{"3":{"4":[5,{"6":7}]}}}
{}
[5]
[5,2]
```

## 7.3.6. PERSISTENCIA DE OBJETOS CON JPA.

Cualquier aplicación de tipo enterprise realiza operaciones con bases de datos para almacenar y recuperar grandes cantidades de información. Pese a todas las tecnologías disponibles para gestionar el

almacenamiento, los desarrolladores de aplicaciones tienen dificultades para realizar operaciones con las bases de datos de forma eficiente.

Normalmente los desarrolladores de Java utilizan Frameworks propietarios o gran cantidad de código para interactuar con las BD, mientras que al usar JPA, las interacciones innecesarias se reducen. JPA es un puente entre el modelo orientado a objetos (aplicaciones de Java) y el modelo relacional (servidores de bases de datos).

**Diferencias entre el modelo relacional y el orientado a objetos.** En el modelo relacional los datos se representan mediante tablas, en la orientación a objetos hay un grafo de interconexiones. Cuando almacenas y recuperas datos en/desde el modelo relacional pueden ocurrir algunos conflictos:

- **Granularidad**: los objetos son más granulares.
- **Subtipos**: significa herencia, no lo soportan todas las BD.
- **Identidad**: como del modelo orientado a objetos, el modelo relacional la implementa de forma diferente.
- **Asociaciones**: el modelo relacional no soporta las relaciones muchos a muchos que si pueden tener los objetos.
- **Navegación de datos**: diferente en ambos modelos.

JPA (**Java Persistence API**) es una colección de clases y métodos para almacenar de forma persistente grandes cantidades de datos en BD.

**Historia de JPA**

En versiones anteriores de EJB, se definía la persistencia combinada con la lógica empresarial usando la interfaz **javax.ejb.EntityBean**.
Pero la cosa fue evolucionando:

- Al introducir EJB 3.0, la capa de persistencia se separó y se especificó como JPA 1.0 (Java Persistence API). Las

especificaciones de la API fueron liberadas en 2006 con Java EE5.

- JPA 2.0 salió con JAVA EE6 en 2009.
- JPA 2.1 salió con JAVA EE7 en 2013.

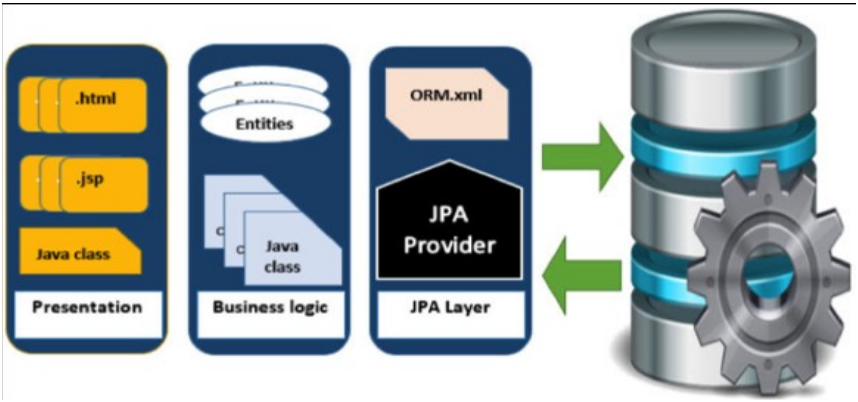



*Figura 8: Arquitectura de una aplicación Enterprise con JPA.*

**Proveedores de JPA**

JPA es una API Open Source, así que los fabricantes de productos empresariales como Oracle, Redhat, Eclipse, etc. cuando crean nuevos productos, también la implementan en ellos. Por ejemplo: Hibernate, Eclipselink, Toplink, Spring Data JPA, etc.

## 7.4. LA API NIO2.

Java 7 introdujo la API NIO con la intención de sustituir a los streams de **java.io** y actualmente NIO2 es una alternativa a **java.io.File**. Intenta aportar una forma más intuitiva y mejorada de trabajar con ficheros.

### LA INTERFACE Path

La interface **java.nio.file.Path** representa una ruta jerárquica del

sistema de archivos que identifica un fichero o un directorio y equivale a la clase **java.io.File**.

### Crear Paths

Como es una interface, necesitas una clase factoria como **java.nio2.Paths** de NIO2 (una clase factoría que se llama **Paths**) que los crean.

**EJEMPLO 30**: Crear varios Path con ayuda de la clase Paths.

```
Path p1 = Paths.get("pandas/miso.png");
Path p2 = Paths.get( "c:\\zoo\\Noviembre\\empleados.txt");
Path p3 = Paths.get("/home/zoodirector");
Path p4 = Paths.get("pandas","miso.png");
Path p5 = Paths.get("c:","zooinfo","Noviembre","empleados.txt ");
Path p6 = Paths.get("/","home","zoodirector ");
```

Otra forma de crearlos es usando la clase **java.net.URI**. Un *uniform resource identifier* (URI) es un string que identifica un recurso en una internet, comienza por un protocolo, le sigue una ruta y acaba con el nombre de un recurso. Por ejemplo, los protocolos pueden ser entre otros: file://, http://, https:// y ftp://.

**EJEMPLO 31**: usar URI y Path.

```
Path p1 = Paths.get( new URI("file://pandas/cuddly.png")); // EXCEPTION
Path p2 = Paths.get( new URI("file:///c:/zoo/noviembre/empleados.txt") );
Path p3 = Paths.get( new URI("file:///home/zoodirector") );
Path p4 = Paths.get( new URI("http://www.gva.es") );
URI uri4 = p4.toUri();
```

## ACCEDER AL SISTEMA DE FICHEROS

El método **Path.getPath()** es un acceso rápido a la clase **java.nio.file.FileSystem.getPath()**. La clase **FileSystem** tiene el contructor protegido, así que usamos la clase factoría **FileSystems**.

**EJEMPLO 32**: acceder a objetos del sistema de ficheros:

```
Path p1 = FileSystems.getDefault().getPath("pandas/cuddly.png ");
```

```
Path p2 = FileSystems.getDefault().getPath("c:","Nov","carta.txt ");
Path p3 = FileSystems.getDefault().getPath("/home/zoodirector");
```

### CAMBIAR ENTRE io Y nio

Usamos los métodos **java.io.File.toPath()** y **java.nio2.Path.toFile()**.

**EJEMPLO 33**: intercambiar entre `File` de io y `Path` de nio.

```
File f = new File("pandas/cuddly.png");
Path p = f.toPath();
p = Paths.get("cuddly.png");
f = path.toFile();
```

### LEER Y ESCRIBIR EN FICHEROS

El método **Files.newBufferedReader(Path, Charset)**, lee el fichero usando un objeto **java.io.BufferedReader**. Puedes usar **Charset.defaultCharset()**.

**EJEMPLO 34**: leer en ficheros usando nio.

```
Path p = Paths.get("/animales/gopher.txt");
try {
   BufferedReader br = Files.newBufferedReader(p,
                              Charset.forName("US-ASCII"));
   // Leer
   String lineaActual = null;
   while( (lineaActual = br.readLine()) != null )
       System.out.println( lineaActual );
} catch (IOException e) {
    // manejar file I/O exception...
}
```

Para escribir podemos usar **Files.newBufferedWriter(Path,Charset)**. Ej:

**EJEMPLO 35**: escribir en ficheros usando nio.

```
Path p = Paths.get("/animals/gorila.txt");
List<String> datos = new ArrayList();
try {
    BufferedWriter bw = Files.newBufferedWriter(p,
                                        Charset.defaultCharset() );
    bw.write("Hola mundo");
```

```
} catch (IOException e) {
    // manejar file I/O exception...
}
```

El método **Files.readAllLines()** permite leer todas las líneas de un fichero de texto y almacenarlas en una lista ordenada de strings. La API NIO.2 sobrecarga esta operación incluyendo un Charset.

**EJEMPLO 36**: leer todas las líneas del fichero.

```
Path p = Paths.get("/fish/sharks.log");
try {
    final List<String> lineas = Files.readAllLines(p);
    for(String linea: lineas) {
        System.out.println(linea);
}
catch (IOException e) {
    // maneja file I/O exception...
}
```

## MANIPULAR FICHEROS CON PROGRAMACIÓN FUNCIONAL

Para procesar datos de manera funcional, disponemos de una serie de operaciones de alto nivel que podemos encadenar de forma que la siguiente operación usa los resultados que genera la anterior.

Estas operaciones trabajan con streams de datos (una colección de datos). Por ejemplo, el método **walk()** crea un stream de paths (rutas a ficheros), incluyendo los ficheros que hay dentro de uno que sea una carpeta. Este ejemplo imprimirá los ficheros .java de un directorio:

```
Path p = Paths.get("/software");
try {
    Files.walk(p)
         .filter( p -> p.toString().endsWith(".java") )
         .forEach( System.out::println );
} catch (IOException e) {
    // maneja file I/O exception...
}
```

Ahora además filtramos por una fecha de modificación y solo queremos los 10 primeros. Usamos el método **find()** en vez de `walk()`:

```
Path p1 = Paths.get("/felinos");
long limite = 1420070400000L;
try {
    Stream<Path> s = Files.find(p1, 10,
           (p,a) -> p.toString().endsWith(".java")
                    && a.lastModifiedTime().toMillis() >  limite);
    stream.forEach(System.out::println);
} catch (Exception e) {
    // maneja file I/O exception...
}
```

**EJEMPLO 37**: para listar los ficheros de un directorio, sin buscar en las subcarpetas como hace `walk()` podemos usar **`Files.list()`**:

```
try {
    Path p1 = Paths.get("patos");
    Files.list(p1)
         .filter( p -> !Files.isDirectory(p) )
         .map( p -> p.toAbsolutePath() )
         .forEach( System.out::println );
} catch (IOException e) {
    // manejar file I/O exception...
}
```

## PROCESAR FICHEROS LARGOS

Si usamos **`Files.readAllLines()`** con ficheros muy grandes podríamos acabar teniendo la excepción **`OutOfMemoryError`**. La API `NIO.2` tiene el método **`Files.lines(Path)`** que procesa el fichero en forma de **`Stream<String>`** y no necesita cargarlo completamente en memoria para trabajar con sus datos. Ejemplo:

```
Path p1 = Paths.get("/peces/tiburones.log");
try {
    Files.lines(p1).forEach( System.out::println );
} catch (IOException e) {
    // manejar file I/O exception...
}
```

La misma ventaja tienen el resto de operaciones de la API streams:

```
Path p1 = Paths.get("/peces/tiburones.log");
try {
    System.out.println(
        Files.lines(path)
             .filter( s -> s.startsWith("WARN") )
             .map( s -> s.substring(5) )
             .collect( Collectors.toList() )
    );
} catch (IOException e) {
    // manejar file I/O exception...
}
```

**Tabla:** Equivalencias de algunos métodos de File y NIO.2

| **Método de File** | **Método de NIO.2** |
|---|---|
| file.exists() | Files.exists(path) |
| file.getName() | path.getFileName() |
| file.getAbsolutePath() | path.toAbsolutePath() |
| file.isDirectory() | Files.isDirectory(path) |
| file.isFile() | Files.isRegularFile(path) |
| file.isHidden() | Files.isHidden(path) |
| file.length() | Files.size(path) |
| file.lastModified() | Files.getLastModifiedTime(path) |
| file.setLastModified(time) | Files.setLastModifiedTime(path,fileTime) |
| file.delete() | Files.delete(path) |
| file.renameTo(otherFile) | Files.move(path,otherPath) |
| file.mkdir() | Files.createDirectory(path) |
| file.mkdirs() | Files.createDirectories(path) |
| file.listFiles() | Files.list(path) |

# 7.5 EJERCICIOS.

## EJERCICIO 1. TEST

**T1.** Señala la opción correcta:

( ) Read es una clase de System que permite leer caracteres.
( ) StringBuffer permite leer y StringBuilder escribir en la salida estándar.
( ) La clase Keyboard también permite leer flujos de teclado.
( ) Stderr por defecto dirige al monitor pero se puede direccionar a otro dispositivo.

**T2.** Indica si es verdadera o falsa la siguiente afirmación: Para flujos de caracteres es mejor usar las clases Reader y Writer en vez de InputStream y OutputStream. **( )Verdadero. ( )Falso.**

**T3.** Indica si es verdadera o falsa la siguiente afirmación: Cuando trabajamos con ficheros en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas. ¿Verdadero o Falso? **( )Verdadero. ( )Falso.**

**T4.** Señala si es verdadera o es falsa la siguiente afirmación: La idea de usar buffers con los ficheros es incrementar los accesos físicos al disco. **( )Verdadero. ( )Falso.**

**T5.** Señala si es verdadera o falsa la siguiente afirmación: Para leer datos desde un fichero codificado en binario empleamos la clase FileOutputStream. **( )Verdadero. ( )Falso**.

**T6.** Señala la opción correcta:

- Java sólo admite el uso de ficheros aleatorios.
- Con los ficheros de acceso aleatorio se puede acceder a un registro determinado directamente.
- Los ficheros secuenciales se deben leer de tres en tres registros.
- Todas son falsas.

**T7.** Señala si es verdadera o es falsa la siguiente afirmación: para encontrar una información almacenada en la mitad de un fichero secuencial, podemos acceder directamente a esa posición sin pasar por los datos anteriores a esa información. **( )Verdadero. ( )Falso.**

**T8.** Indica si es verdadera o es falsa la siguiente afirmación: Para decirle el modo de lectura y escritura a un objeto RandomAccessFile debemos pasar como parámetro "rw". **( )Verdadero. ( )Falso.**

**T9.** Un objeto de la clase File representa un fichero en sí mismo. ¿Verdadero o falso? ( )Verdadero. ( )Falso.

**T10.** Una clase que implemente **FileNameFilter** puede o no implementar el método accept. ( )Verdadero. ( ) Falso.

**T11.** Indica cuál de los siguientes streams se puede utilizar en Java para leer datos desde un origen: fichero, array, periférico o socket.

a) InputStream
b) OutputStream
c) Input/OutputStream
d) Ninguno de los anteriores

**T12.** Estos métodos: read(), available() y close(). ¿cuál de los siguientes streams los usa?

a) OutputStream
b) InputStream
c) Input/OutputStream
d) Ninguno de los anteriores

**T13.** Indica qué stream usa un buffer interno y aporta más eficiencia que escribir directamente en un stream de salida. Así que ¿Cuál mejora el rendmiento de salida?

a) BufferedOutputStream
b) ByteArrayOutputStream
c) BufferedInputStream
d) ByteArrayInputStream

**T14**. ¿Cuales de estos elementos permiten leer datos desde el teclado?

a) InputStreamReader
b) Console
c) Scanner
d) DataInputStream
e) Todos los anteriores.

**T15**. ¿Qué clase puede utilizarse para leer datos línea a línea con readLine()?

a) BufferedReader
b) InputStreamReader
c) DataInputStream
d) Ninguno de los anteriores.

**EA2**. Pregunta por el nombre de cualquier archivo existente (debe dar igual si es de texto o binario). Comprueba si existe y no es un directorio, indicando esas circunstancias si ocurren y no haciendo nada más. Si no ocurre ninguna de estas dos situaciones, usa dos streams que te permitan copiar los datos del fichero original en otro fichero llamado "nombre_del_anterior" + "_copia" + ".extensión_del_anterior" usando operaciones read() y write() de los streams.

**EA3**. Modifica el ejercicio EA2 para medir el tiempo que tarda en copiar cualquier fichero y lo pruebas con alguno grande (que pese más de 1MiB).

**EA4**. Repite el ejercicio anterior EA3 pero define un buffer (un array de buffers) de 1024 bytes y cambia las operaciones read() y write(int byte) por **read(buffer)** y **write(buffer)**. ¿Qué le pasa al tiempo de procesamiento al copiar archivos grandes usando buffers?

**EA5.** Repite el ejercicio anterior EA4 pero en vez de definir y usar directamente un buffer, envuelve el stream en otro que utilice buffers. ¿Se resiente significativamente el tiempo por el hecho de no manejar tu directamente el buffer?

**EA6**. Crea un programa EA6.java que acepte de parámetro la ruta a una carpeta y una extensión de ficheros. Si llegan los parámetros llama al método **muestraFicheros(String carpeta, String extensión)** y pasas tanto el directorio como la extensión. El método muestraFicheros() debe:

- Comprobar si existe el fichero d y si no existe, indicarlo y salir.
- Comprobar que sea un directorio y si no lo es, indicarlo y salir.
- Crea una clase anónima que instancie un FileFilter que sobreescriba su método accept() y que devuelva true si el nombre del fichero que recibe acaba en la extensión indicada por el segundo argumento.
- Luego imprime por consola todos los nombres de los ficheros del directorio que pasan el filtro.

**EA7**. Modifica el ejercicio anterior y crea EA7.java para usar un FilenameFilter en vez de un FileFilter y en vez de listar los ficheros, haz que los borre y muestre por pantalla los nombres de los que borra.

**EA8**. Cambia la forma de crear el FilenameFilter del ejercicio anterior y usa una expresión lambda en el ejercicio EA8.java.

**EA9**. Crea ahora el programa EA9.java que acepte de parámetro externo el nombre de una carpeta y si no se indica o no existe o no es

una carpeta, use la carpeta actual '.' como directorio de trabajo. Por último crea un FilenameFilter con una expresión regular que devuelva true para todos los ficheros de esa carpeta cuyos nombres tengan solamente letras.

**EA10**. Crea el programa EA10.java que lee un fichero .CSV llamado "animales.csv" donde unos campos se separan de otros con el carácter dos puntos ':' y tiene líneas con esta estructura: animal:tipo:lugar pero el lugar es opcional (puedes escribirlo tu mismo con el bloc de notas o copiarlo) y este es su contenido:

```
Luisa:Gorila
Manolo:Cocodrilo
Sebas:Tigre:Bengala
```

Para leer sus campos usa un Scanner al que indicas el delimitador y como hay líneas que tienen un número de campos diferentes, usa hasNext() para comprobar si has llegado al final del fichero. Simplemente, imprime cada campo separado por el carácter '|'.

**EA11**. Crea el programa EA11.java que lea un fichero de texto llamado test.txt línea a línea: crea el fichero, lo usas como origen de un FileInputStream, al que envuelves con un BufferedReader, y construye una línea de texto en un StringBuilder para finalmente convertirlo a String y mostrarlo por la consola.

**EA12**. Haz el programa EA12.java que haga algo parecido al anterior, pero ahora usa un InputStreamReader que envuelva a un FileInputStream y deje los char que lea con read() en un array de char de 50 de tamaño, solo realiza una operación de lectura (esto tiene sentido hacerlo para ficheros pequeños donde todos sus caracteres puedan caber en el array). Luego imprime el array.

**EA13**. Haz el programa EA13.java que lea un fichero de texto con un BufferedReader que envuelva a un FileReader y lea línea a línea e imprima la longitud de cada una y cuantas líneas tiene.

**EA14**. La clase InputStreamReader se usa para leer datos de un stream de bytes. Envolviendo a InputStreamReader con un BufferedReader aumentarás la eficiencia. Crea el programa EA14.java que lea la entrada de consola del usuario (System.in) usando un BufferedReader que envuelve un InputStreamReader y pregunta al usuario su nombre y lees la línea desde el BufferedReader. Luego imprime "Bienvenido " + nombre.

**EA15**. Crea el programa EA15.java que genere un fichero de texto temporal con File.createTempFile(String prefijo, String extension). Luego imprime la ruta absoluta que señala al fichero creado. Responde a estas preguntas:

a) ¿Puedes indicar dos letras en el nombre?
b) ¿En qué carpeta se crea en Windows? ¿Y en Linux?
c) ¿Cuál es su nombre?

**EA16**. Haz el programa EA16.java que cree un fichero temporal "maniobra.txt" y haga que se borre automáticamente cuando el programa acabe su ejecución con deleteOnExit() de java.io.File. Utiliza Path de nio en vez de File de io para crear el fichero. Imprime la ruta absoluta del fichero creado.

**EA17**. Haz el programa EA17.java que averigue el directorio de trabajo actual y lo imprima de dos maneras diferentes. La primera usando System y la segunda usando Path y Paths de nio.

**EA18**. Haz el programa EA18.java que pregunte por la edad al usuario y lea el dato de tipo int de forma segura desde la consola con un Scanner, asegurándose antes de que el dato esperado es del tipo

correcto y limpiando el buffer e intentándolo otra vez si no lo es. No debe usar excepciones ni parseos.

**EA19**. Haz el programa EA19.java que almacene en el fichero "ea19.txt" una cadena de texto como si fueran bytes en formato binario. Para mover texto, sería mejor un FileWriter. Pero podría ser un array de bytes cualquiera. Pasa el String a array de bytes y usa el método write( array ) del FileOutputStream.

**EA20**. Haz el programa EA20.java que use DataOutputStream para para guardar en un fichero "e20.bin" el nombre y la edad de un usuario que lee desde consola. Se ayudará de un BufferedOutputStream que a su vez envuelve a un FileOutputStream. Luego cierra el flujo y vuelve a leer los datos con un DataInputStream de forma similar y los imprime.

**EA21**. Haz el programa EA21.java que calcule el código hash MD5 de un fichero y luego lo imprima. Para crear el código hash puedes usar el objeto que devuelve la llamada al método MessageDigest.getInstance("MD5"). Está en el paquete java.security y puede generar la excepción NoSuchAlgorithmException). El programa lee datos binarios de un fichero usando un FileInputStream que envuelve un File. Lee el fichero en trozos de 1024 bytes (1Kb) que va almacenando en el buffer de bytes y se los va pasando al objeto digest con el método update(array, 0, bytes_leidos). Luego cierra el fichero y guarda el código en un array de bytes que devuelve el método digest(). Por último convierte estos bytes en formato hexadecimal y los va almacenando en un StringBuilder y lo imprime por consola.

**EA22**. Haz el programa EA22.java que adapte el código del ejercicio anterior y cree el método public String getFileChecksum( String hash, String file) que devuelva el hash MD5 y SHA-256 de un fichero.

**EA23**. Pregunta por un fichero con extensión .BMP y por el nombre de una copia y generas en la copia el mismo fichero pero la imagen transformada a escala de grises. Para hacer esto debes leer el color de cada píxel (debes asegurarte que cada píxel son 24 bits formados por 3 bytes y cada uno con la componente R, G, B).

Puedes coger cualquier imagen, copiarla al paint y guardarla como un .BMP de 24 bits por píxel. Un color gris se caracteriza porque cada componente es igual a las demás, es decir la cantidad de rojo es igual a la de verde e igual a la de azul (R = G = B). Por tanto, si quieres transformar un pixel a color en su version en escala de grises debes leer cada componente del pixel y sacarle la media:

$$\text{media} = \frac{R + G + B}{3}$$

Y en el fichero de destino no guardas R ni G ni B sino la media obtenida 3 veces: media media media

El último problema que tienes que solucionar es localizar donde están en el fichero los bytes de los píxels de la imagen. Para ello debes conocer el formato de un fichero BMP, que básicamente es:

| 2 | 4 | 4 | 4 | Cabecera de la imagen, paleta... | píxels |
|---|---|---|---|---|---|

Los campos de izquierda a derecha en el fichero:

- 2 bytes con las letras 'B' y 'M'. Es lo que se conoce como firma. Sirve a los programas para asegurarse de que el fichero es realmente lo que se supone que indica su extensión (Un usuario podría cambiar el nombre del fichero `carta.txt` por `foto.bmp`).
- 4 bytes con el peso del archivo en bytes.
- 4 bytes para uso reservado.
- 4 bytes que contienen el `offset`, que es el desplazamiento donde

empiezan los bytes con la información de los píxels de la imagen con respecto del comienzo del fichero. En rojo en el diagrama indica la posición de la información de los píxels. Si la imagen es de 24 bits por píxel (3 bytes), no usa paleta de colores, ni ningún método de compresión, al leer 3 bytes desde esa posición, lees el color RGB de un píxel.

**EA24.** Haz el programa EA24.java que procese el fichero empleados.xml para extraer su información e imprimirla por consola:

```
<empleados>
<empleado id="111">
<nombre>Luis</nombre>
<apellido>Garci</apellido>
<lugar>Alicante</lugar>
</empleado>
<empleado id="222">
<nombre>Alex</nombre>
<apellido>Iglesia</apellido>
<lugar>Valencia</lugar>
</empleado>
<empleado id="333">
<nombre>David</nombre>
<apellido>Busta</apellido>
<lugar>Madrid</lugar>
</empleado>
</empleados>
```

**EA25**. Haz el programa EA25.java que liste por consola los ficheros que hay que contiene el fichero "e25.zip" que habrás creado antes con un par de ficheros dentro y los extraiga en la carpeta actual.

**EA26**. Haz el programa EA26.java que use un JFileChooser para leer una imagen (activa la carpeta actual como su carpeta de inicio y un filtro de extensión para que solo vea ficheros .jpg, .jpeg, .png, .gif y .bmp).
Si el usuario escoge una imagen, utiliza la clase ImageIO de

javax.swing para leerla con read(File) en un BufferedImage. Esta clase te permite manipular sus pixels. Puedes averiguar cuantos pixels tiene de alto con getHeight() y puedes saber cuantos de ancho con getWidth(). Además puedes acceder al valor de cada uno con **int getRGB(columna, fila)** que devuelve un entero con el color que guarda. El color está codificado en un int como:

TTTTTTTTRRRRRRRRGGGGGGGGBBBBBBB
- T -> 8 bits para indicar transparencia
- R -> 8 bits para indicar cantidad de rojo del color (de 0 a 255)
- G -> 8 bits para indicar cantidad de verde.
- B -> 8 bits para indicar cantidad de azul.

También puedes cambiar un píxel con **setRGB(int columna, int fila, int color)**. Sabiendo que un color gris tiene la característica de que los 3 componentes RGB tienen el mismo valor:

(0,0,0) -> negro
(1, 1, 1) -> gris muuuuy oscuro
(2, 2, 2) -> gris muuy negro
:
(254,254,254) -> gris casi blanco
(255,255,255) -> blanco

Y que de cualquier color RGB con los valores (x,y,z) puedes calcular su equivalente en escala de grises como:

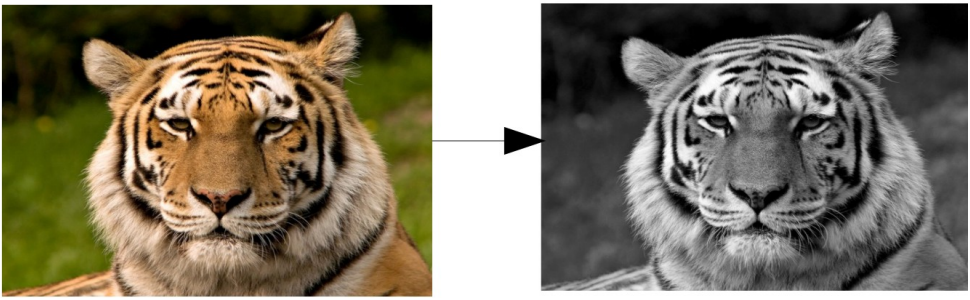
media <- ( x + y + z) / 3

equivalente:

(media, media, media)

Transforma la imagen que leas en su versión en escala de grises y vuelve a usar **ImageIO** para guardarla de nuevo con el formato que

quieras usando de nuevo el `JFileChooser` (puedes aprovechar el que tenías antes).



**EA27.** Haz el programa `EA27.java` que usando la clase `java.awt.Robot` nos permita crear una imagen de un trozo de la pantalla con su método `createScreenCapture(Rectangle r)`, que devuelve una `BufferedImage`. En nuestro caso vamos a averiguar el tamaño de toda la pantalla con la clase `java.awt.Toolkit` que tiene un método estático llamado `getDefaultToolkit().getScreenSize()` que devuelve un `Rectangle`. Por último haremos que `ImageIO` nos guarde la imagen como un `jpg` de nombre los milisegundos de la hora actual del sistema y en la carpeta usada actualmente por el usuario.

**EA28.** Haz un programa que muestre este menú:

1. Leer notas de fichero.
2. Añadir/Borrar/Modificar notas.
3. Ver notas y calcular la media.
4. Gardar notas en fichero.
5. Salir.

Debe implementar las opciones usando streams de texto con el fichero `notas.csv` con el formato: `MODULO;nota` El programa no debe permitir que haya más de una línea con el mismo módulo.

**EA29**. Repite el programa anterior pero usando un fichero binario llamado notas.bin. (Puedes ayudarte de colecciones).