

Lectura y escritura de Información en Java.



Objetivos

Esta unidad de trabajo del módulo trata sobre la generación de aplicaciones de entorno Java con operaciones de Entrada y Salida.

En ella se tratarán los flujos, ficheros binarios, los de texto y los accesos secuencial y aleatorio a ficheros y la serialización de datos.

En esta unidad de trabajo se pretende que el alumnado alcance los siguientes objetivos:

- ✓ Realizar operaciones de entrada y salida de información, utilizando procedimientos específicos del lenguaje y librerías de clases.
- ✓ Gestionar la información almacenada para desarrollar aplicaciones de gestión.
- ✓ Emplear herramientas de desarrollo, y operaciones del lenguaje, siguiendo las especificaciones y verificando interactividad y usabilidad, para desarrollar aplicaciones multiplataforma.
- ✓ Verificar las aplicaciones software desarrolladas, analizando las especificaciones.

1.- Introducción.

Cuando desarrollas programas, en la mayoría de ellos los usuarios pueden pedirle a la aplicación que realice cosas y pueda suministrarte datos con los que se quiere hacer algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos, para proporcionar una respuesta a lo solicitado.

Además, normalmente interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros, que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

Por tanto, sabemos que el almacenamiento en variables o vectores (arrays) es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o cuando el programa termina. **Las computadoras utilizan ficheros para guardar los datos**, incluso después de que el programa termine su ejecución. Se suele denominar a los datos que se guardan en ficheros **datos persistentes**, porque existen, persisten más allá de la ejecución de la aplicación. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos cómo hacer con Java estas operaciones de crear, actualizar y procesar ficheros.

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como **Entrada/Salida (E/S)**.

Distinguimos dos tipos de E/S: la **E/S estándar** que se realiza con el terminal del usuario y la **E/S a través de ficheros**, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar del API de Java denominado `java.io` que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

El contenido de un archivo puede interpretarse como **campos** y **registros** (grupos de campos), dándole un significado al conjunto de bits que en realidad posee.

1.1.- Excepciones.

Cuando se trabaja con archivos, es normal que pueda haber errores, por ejemplo: podríamos intentar leer un archivo que no existe, o podríamos intentar escribir en un archivo para el que no tenemos permisos de escritura. Para manejar todos estos errores debemos utilizar excepciones. Las dos excepciones más comunes al manejar archivos son:

- ✓ **FileNotFoundException**: Si no se puede encontrar el archivo.
- ✓ **IOException**: Si no se tienen permisos de lectura o escritura o si el archivo está dañado.

Un esquema básico de uso de la captura y tratamiento de excepciones en un programa, podría ser este, importando el paquete `java.io.IOException`:

```
import java.io.IOException;

public class prueba1 {
    // ...

    public static void main(String[] args) {
        try {
            // Se ejecuta algo que puede producir una excepción
            } catch (FileNotFoundException e) {
                // manejo de una excepción por no encontrar un archivo
            } catch (IOException e) {
                // manejo de una excepción de entrada/salida
            } catch (Exception e) {
                // manejo de una excepción cualquiera
            } finally {
                // código a ejecutar haya o no excepción
            }
        }
    }
}
```



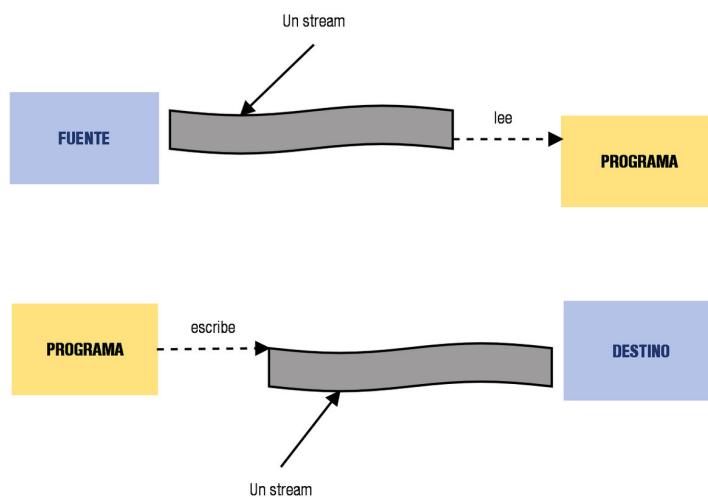
Autoevaluación

Señala la opción correcta:

- Java no ofrece soporte para excepciones.
- Un campo y un archivo es lo mismo.
- Si se intenta abrir un archivo que no existe, entonces saltará una excepción.
- Ninguna es correcta.

2.- Concepto de flujo.

La clase **Stream** representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida (en adelante E/S), memoria, un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc.



Cualquier programa realizado en Java que necesite llevar a cabo una operación de entrada salida lo hará a través de un **stream**.

Un flujo es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

Las clases y métodos de E/S que necesitamos emplear son las mismas **independientemente del dispositivo con el que estemos actuando**, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red liberando al programador de tener que saber con quién está interactuando.

La vinculación de un flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

En resumen, será el flujo el que tenga que comunicarse con el sistema operativo concreto y "entendérselas" con él. De esta manera, **no tenemos que cambiar absolutamente nada en nuestra aplicación**, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es primordial en un lenguaje multiplataforma y tan altamente portable como Java.



Autoevaluación

Señala la opción correcta:

- La clase Stream puede representar, al instanciarse, a un archivo.
- Si programamos en Java, hay que tener en cuenta el sistema operativo cuando tratemos con flujos, pues varía su tratamiento debido a la diferencia de plataformas.
- La clase keyboard es la clase a utilizar al leer flujos de teclado.
- La vinculación de un flujo al dispositivo físico la hace el hardware de la máquina.

3.- Clases relativas a flujos.

Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (character streams).

- ✓ Los **flujos de caracteres** (16 bits) se usan para manipular datos legibles por humanos (por ejemplo un fichero de texto). Viene determinados por dos clases abstractas: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres UNICODE. De ellas derivan subclases concretas que implementan los métodos definidos destacados los métodos **read()** y **write()** que, en este caso, leen y escriben **caracteres** de datos respectivamente.
- ✓ Los **flujos de bytes** (8 bits) se usan para manipular datos binarios, legibles solo por la máquina (por ejemplo un fichero de programa). Su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son **InputStream** y **OutputStream**. Estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan **read()** y **write()** que leen y escriben bytes de datos respectivamente.

Las clases del paquete `java.io` relativas a flujos son:

- ✓ **BufferedInputStream**: permite leer datos a través de un flujo con un buffer intermedio.
- ✓ **BufferedOutputStream**: implementa los métodos para escribir en un flujo a través de un buffer.
- ✓ **FileInputStream**: permite leer bytes de un fichero.
- ✓ **FileOutputStream**: permite escribir bytes en un fichero o descriptor.
- ✓ **StreamTokenizer**: esta clase recibe un flujo de entrada, lo analiza (parse) y divide en diversos pedazos (tokens), permitiendo leer uno en cada momento.
- ✓ **StringReader**: es un flujo de caracteres cuya fuente es una cadena de caracteres o string.
- ✓ **StringWriter**: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.

Destacar que hay clases que se **"montan"** sobre otros flujos para modificar la forma de trabajar con ellos. Por ejemplo, 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.

3.1.- Ejemplo comentado de una clase con flujos.

Vamos a ver un ejemplo con una de las clases comentadas, en concreto, con StreamTokenizer.

La clase **StreamTokenizer** obtiene un flujo de entrada y lo divide en "tokens". El flujo tokenizer puede reconocer identificadores, números y otras cadenas.

El ejemplo siguiente, muestra cómo utilizar la clase **StreamTokenizer** para contar números y palabras de un fichero de texto. Se abre el flujo con ayuda de la clase **FileReader**, y puedes ver cómo se "monta" el flujo **StreamTokenizer** sobre el **FileReader**, es decir, que se construye el objeto **StreamTokenizer** con el flujo **FileReader** como argumento, y entonces se empieza a iterar sobre él.

```

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

public class token {

    public void contarPalabryNumeros(String nombre_fichero) {

        StreamTokenizer sTokenizer = null;
        int contapal = 0, numNumeros = 0;

        try {
            sTokenizer = new StreamTokenizer(new FileReader(nombre_fichero));
            while (sTokenizer.nextToken() != StreamTokenizer.TT_EOF) {

                if (sTokenizer.ttype == StreamTokenizer.TT_WORD)
                    contapal++;
                else if (sTokenizer.ttype == StreamTokenizer.TT_NUMBER)
                    numNumeros++;
            }

            System.out.println("Número de palabras en el fichero: " + contapal);
            System.out.println("Número de números en el fichero: " +
numNumeros);

        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }

    public static void main(String[] args) {
        new token().contarPalabryNumeros("c:\\ejerjava\\datos.txt");
    }
}

```

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

En el código de la clase, apreciamos que se iterará 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.



Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa.

Según el sistema operativo que utilicemos, habrá que utilizar un flujo u otro. ¿Verdadero o Falso?

- Verdadero
- Falso

4.- Flujos.

Hemos visto qué es un flujo y que existe un árbol de clases amplio para su manejo. Ahora vamos a ver en primer lugar los **flujos predefinidos**, también conocidos como de entrada y salida, y después veremos los **flujos basados en bytes** y los **flujos basados en carácter**.

4.1.- Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente, los usuarios del sistema operativo Unix, Linux y también MS-DOS, han utilizado un tipo de entrada/salida conocida comúnmente por 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` 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 de la entrada estándar:

- ✓ `int System.in.read();`
 - ➡ Lee el siguiente `byte` (`char`) de la entrada estándar.
- ✓ `int System.in.read(byte[] b);`
 - ➡ Leer un conjunto de bytes de la entrada estándar y lo almacena en el array `b`.

Para la salida, se usa el método `print` para escribir en la salida estándar:

- ✓ `System.out.print(String);`
 - ➡ Muestra el texto en la consola.
- ✓ `System.out.println(String);`
 - ➡ Muestra el texto en la consola y seguidamente 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.

Funciones de conversión.

Método	Funcionamiento
<code>byte Byte.parseByte(String)</code>	Convierte una cadena en un número entero de un byte
<code>short Short.parseShort(String)</code>	Convierte una cadena en un número entero corto
<code>int Integer.parseInt(String)</code>	Convierte una cadena en un número entero
<code>long Long.parseLong(String)</code>	Convierte una cadena en un número entero largo
<code>float Float.parseFloat(String)</code>	Convierte una cadena en un número real simple
<code>double Double.parseDouble(String)</code>	Convierte una cadena en un número real doble
<code>boolean Boolean.parseBoolean(String)</code>	Convierte una cadena en un valor lógico

4.2.- Flujos predefinidos. Entrada y salida estándar. Ejemplo.

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.

Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase `StringBuffer` o la `StringBuilder`. La clase `StringBuffer` permite almacenar cadenas que cambiarán en 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`.

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 iremos almacenando los caracteres que se escriban
        StringBuilder str = new StringBuilder();
        char c;
        // Por si ocurre una excepción ponemos el bloque try-catch
        try{
            // Mientras la entrada de teclado no sea Intro
            while ((c=(char)System.in.read())!='\n'){
                // Añadir el carácter leído a la cadena str
                str.append(c);
            }
        }catch(IOException ex){
            System.out.println(ex.getMessage());
        }
        // Escribir la cadena que se ha ido tecleando
        System.out.println("Cadena introducida: " + str);
    }
}

```



Autoevaluación

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 direccional a otro dispositivo.

4.3.- Flujos basados en bytes.

Este tipo de flujos 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 que son **InputStream** y **OutputStream**. Cada una de estas clases abstractas tiene varias subclases concretas, 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 y el **InputStream** y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```

void copia (String origen, String destino) throws IOException {
    try{
        // Obtener los nombres de los ficheros de origen y destino
        // y abrir la conexión a los ficheros.
        InputStream fentrada = new FileInputStream(origen);
        OutputStream fsalida = new FileOutputStream(destino);
        // Crear una variable para leer el flujo de bytes del origen
        byte[] buffer= new byte[256];
        while (true) {
            // Leer el flujo de bytes
            int n = fentrada.read(buffer);
            // Si no queda nada por leer, salir del while
            if (n < 0)
                break;
            // Escribir el flujo de bytes leídos al fichero destino
            fsalida.write(buffer, 0, n);
        }
        // Cerrar los ficheros
        fentrada.close();
        fsalida.close();
    }catch(IOException ex){
        System.out.println(ex.getMessage());
    }
}

```

4.4.- Flujos basados en caracteres.

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 con **caracteres Unicode**, los cuales están **representados por dos bytes**. Por eso, se consideró necesaria la creación de las clases orientadas al flujo de caracteres para ofrecernos el soporte necesario para el tratamiento de caracteres.

Para los flujos de caracteres, Java dispone de dos clases abstractas: **Reader** y **Writer**. Éstas y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

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 puede ser envuelto por otro flujo para tratar el flujo de datos de forma cómoda. Así, un **bufferWriter** nos permite manipular el flujo de datos como un buffer, pero si lo envolvemos en un **PrintWriter** lo podemos escribir con muchas más funcionalidades adicionales para diferentes tipos de datos.

En este ejemplo de código, se ve cómo podemos escribir la salida estándar a un fichero. Cuando se teclee la palabra "salir", se dejará de leer y entonces se saldrá del bucle de lectura.

Podemos ver cómo se usa **InputStreamReader** que es un puente de flujos de bytes a flujos de 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, vectores y líneas.

Como vemos en el código, usamos **FileWriter** para flujos de caracteres, pues para datos binarios se utiliza **FileOutputStream**.

```
try{
    PrintWriter out = null;
    out = new PrintWriter(new FileWriter("c:\\salida.txt", true));

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String s;
    while (!(s = br.readLine()).equals("salir")){
        out.println(s);
    }
    out.close();
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```



Autoevaluación

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

4.5.- Rutas de los ficheros.

En los ejemplos que vemos en el tema estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:



Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo, y por tanto persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: `File.separator`.

Podríamos hacer una función que al pasarle una ruta nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
String substFileSeparator(String ruta){
    String separador = "\\";
    try{
        // Si estamos en Windows
        if ( File.separator.equals(separador) )
            separador = "/";
        // Reemplaza todas las cadenas que coinciden con la expresión
        // regular dada oldSep por la cadena File.separator
        return ruta.replaceAll(separador, File.separator);
    }catch(Exception e){
        // Por si ocurre una java.util.regex.PatternSyntaxException
        return ruta.replaceAll(separador + separador, File.separator);
    }
}
```



Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación.

Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas. ¿Verdadero o Falso?

- Verdadero Falso

5.- Trabajando con ficheros.

En este apartado vas a ver muchas cosas sobre los ficheros: cómo leer y escribir en ellos, aunque ya hemos visto algo sobre eso, hablaremos de las formas de acceso a los ficheros: secuencial o de manera aleatoria.

Siempre hemos de tener en cuenta que la manera de proceder con ficheros debe ser:

- ✓ **Abrir** o bien **crear** si no existe el fichero.
- ✓ **Hacer las operaciones** que necesitemos.
- ✓ **Cerrar el fichero**, para no perder la información que se haya modificado o añadido.

También es muy importante el **control de las excepciones**, para evitar que se produzcan fallos en tiempo de ejecución. Si intentamos abrir sin más un fichero, sin comprobar si existe o no, y no existe, saltará una excepción.



Para saber más

En el siguiente enlace a la wikipedia podrás ver la descripción de varias extensiones que pueden presentar los archivos.

[Extensión de un archivo.](#)

5.1.- Escritura y lectura de información en ficheros.

Acabamos de mencionar los pasos fundamentales para proceder con ficheros: abrir, operar, cerrar.

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, es más eficiente emplear las clases de **Reader Writer**, frente a las clases de **InputStream** y **OutputStream** que están indicadas para flujos de bytes.

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**. Sin embargo, en un fichero de acceso aleatorio podemos posicionarnos directamente en una posición del fichero, y ahí leer o escribir.

Aunque ya has visto un ejemplo que usa **BufferedReader**, insistimos aquí sobre la filosofía de estas clases, que usan la idea de un buffer.

La idea es que cuando una aplicación necesita leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le proporcione la información.

Un dispositivo cualquiera de memoria masiva, por muy rápido que sea, es mucho más lento que la CPU del ordenador.

Así que, es fundamental **reducir el número de accesos al fichero** a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia, el buffer, de modo que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo, ya que es una memoria mucho más rápida que cualquier otro dispositivo de memoria masiva.

Cualquier operación de Entrada/Salida a ficheros puede generar una **IOException**, es decir, un error de Entrada/Salida. Puede ser por ejemplo, que el fichero no exista, o que el dispositivo no funcione correctamente, o que nuestra aplicación no tenga permisos de lectura o escritura sobre el fichero en cuestión. Por eso, las sentencias que involucran operaciones sobre ficheros, deben ir en un bloque **try**.



Autoevaluación

Señala si es verdadera o es falsa la siguiente afirmación:

La idea de usar buffers con los ficheros es incrementar los accesos físicos a disco.
¿Verdadero o falso?

- Verdadero Falso

5.2.- Ficheros de texto.

En los **ficheros de texto** la información se guarda como caracteres. Esos caracteres están codificados en **Unicode**, en **ASCII** u otras codificaciones de texto.

En la siguiente porción de código puedes ver cómo para un fichero existente, que en este caso es texto.txt, averiguamos la codificación que posee, usando el método `getEncoding()`

```
FileInputStream fichero;
try {
    // Elegimos fichero para leer flujos de bytes "crudos"
    fichero = new FileInputStream("c:\\texto.txt");
    // InputStreamReader sirve de puente de flujos de byte a caracteres
    InputStreamReader unReader = new InputStreamReader(fichero);
    // Vemos la codificación actual
    System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {
    Logger.getLogger(textos.class.getName()).log(Level.SEVERE, null, ex);
}
```

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** a partir de **FileReader**. Lo podemos hacer de la siguiente forma:

```
File arch = new File ("C:\\fich.txt");
FileReader fr = new FileReader (arch);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para **escribir en archivos de texto** lo podríamos hacer, teniendo en cuenta:

```
FileWriter fich = null;
PrintWriter pw = null;
fich = new FileWriter("c:/fich2.txt");
pw = new PrintWriter(fichero);
pw.println("Linea de texto");
...
```

Si el fichero al que queremos escribir existe y lo que queremos es añadir información, entonces pasaremos el segundo parámetro como true:

```
FileWriter("c:/fich2.txt",true);
```

5.3.- Ficheros binarios.

Los **ficheros binarios** almacenan la información en bytes, 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. O sea que, cuando se guarda texto no se guarda el texto en sí, sino que se guarda su representación en código UTF-8

Para **leer datos** de un fichero binario, Java proporciona la clase **FileInputStream**. Dicha clase trabaja con bytes que se leen desde el flujo asociado a un fichero.

Para **escribir datos** a un fichero binario, la clase que nos permite usar un fichero para escritura de bytes en él, es la clase **FileOutputStream**. La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria, desde la aplicación que hace de fuente de datos hasta el fichero, que los consume.

Aquí puedes ver un ejemplo comentado.

```

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class LeerConBuffer {

    public static void main(String[] args) {
        int tama ;
        try{
            // Creamos un nuevo objeto File
            File f = new File("C:\\apuntes\\test.bin");

            // Construimos un flujo de tipo FileInputStream (un flujo de entrada desde
            // fichero) sobre el objeto File. Estamos conectando nuestra aplicación
            // a un extremo del flujo, por donde van a salir los datos, y "pidiendo"
            // al SO que conecte el otro extremo al fichero que indica
            // la ruta establecida por el objeto File f creado
            FileInputStream flujoEntrada = new FileInputStream(f);
            BufferedInputStream fEntradaConBuffer = new BufferedInputStream(flujoEntrada);

            // Escribimos el tamaño del fichero en bytes.
            tama = fEntradaConBuffer.available();
            System.out.println("Bytes disponibles: " + tama);

            // Indicamos que vamos a intentar leer 50 bytes del fichero.
            System.out.println("Leyendo 50 bytes....");

            // Creamos un array de 50 bytes para llenarlo con los 50 bytes
            // que leamos del flujo (realmente del fichero)
            byte bytearray[] = new byte[50];

            // El método read() de la clase FileInputStream recibe como parámetro un
            // array de byte, y lo llena leyendo bytes desde el flujo.
            // Devuelve un número entero, que es el número de bytes que realmente se
            // han leido desde el flujo. Si el fichero tiene menos de 50 bytes, no
            // podrá leer los 50 bytes, y escribirá un mensaje indicandolo.
            if (fEntradaConBuffer.read(bytearray) != 50)
                System.out.println("No se pudieron leer 50 bytes");

            // Usamos un constructor adecuado de la clase String, que crea un nuevo
            // String a partir de los bytes leídos desde el flujo, que se almacenaron
            // en el array bytearray, y escribimos ese String.
            System.out.println(new String(bytearray, 0, 50));

            // Finalmente cerramos el flujo. Es importante cerrar los flujos
            // para liberar ese recurso. Al cerrar el flujo, se comprueba que no
            // haya quedado ningún dato en el flujo sin que se haya leído por la aplicación.
            fEntradaConBuffer.close();

            // Capturamos la excepción de Entrada/Salida.
        }catch (IOException e){
            System.err.println("No se encuentra el fichero");
        }
    }
}

```



Autoevaluación

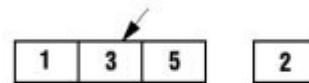
Señala si es verdadera o falsa la siguiente afirmación:

Para leer datos desde un fichero codificados en binario empleamos la clase `FileOutputStream`.
¿Verdadero o falso?

- Verdadero
- Falso

5.4.- Modos de acceso. Registros.

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 los archivos que se crean con Java. Por tanto, los programadores deben estructurar los ficheros de modo que cumplan con los requerimientos de sus aplicaciones.



Así, el programador definirá su registro con el número de bytes que le interesen, moviéndose luego por el fichero teniendo en cuenta ese tamaño que ha definido.

Se dice que un fichero es de acceso directo o de organización directa cuando para acceder a un registro n cualquiera, 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, de manera que para recuperarla se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el registro del fichero que ocupa la posición tres (en la ilustración sería el número 5), tendremos que abrir el fichero y leer los primeros tres registros, hasta que finalmente leamos el registro número tres.

Por el contrario, si se tratara de un **fichero de acceso aleatorio**, podríamos acceder directamente a la posición tres del fichero, o a la que nos interesa.



Autoevaluación

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.

5.5.- 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 `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 a consola.

```

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class LeeYscribe {

    public static void main(String[] args) {
        // Declarar un objeto de tipo archivo
        DataOutputStream archivo = null ;
        DataInputStream fich = null ;
        String nombre = null ;
        int edad = 0 ;
        try {
            // Creando o abriendo para añadir el archivo
            archivo = new DataOutputStream( new FileOutputStream("c:\\ejerjava\\secuencial.dat",true)

                // Escribir el nombre y los apellidos
                archivo.writeUTF( "Antonio Lopez Perez      " );
                archivo.writeInt(33) ;
                archivo.writeUTF( "Pedro Piqueras Peñaranda" );
                archivo.writeInt(45) ;
                archivo.writeUTF( "Jose Antonio Ruiz Perez " ) ;
                archivo.writeInt(51) ;
                // Cerrar fichero
                archivo.close();

                // Abrir para leer
                fich = new DataInputStream( new FileInputStream("c:\\ejerjava\\secuencial.dat") );
                nombre = fich.readUTF() ;
                System.out.println(nombre) ;
                edad = fich.readInt() ;
                System.out.println(edad) ;
                nombre = fich.readUTF() ;
                System.out.println(nombre) ;
                edad = fich.readInt() ;
                System.out.println(edad) ;
                nombre = fich.readUTF() ;
                System.out.println(nombre) ;
                edad = fich.readInt() ;
                System.out.println(edad) ;
                fich.close();

            } catch(FileNotFoundException fnfe) { /* Archivo no encontrado */ }
            catch (IOException ioe) { /* Error al escribir */ }
            catch (Exception e) { /* Error de otro tipo*/}
                System.out.println(e.getMessage());}

    }
}

```

Fíjate al ver el código, que hemos tenido la precaución de ir escribiendo las cadenas de caracteres con el mismo tamaño, de manera que sepamos luego el tamaño del registro que tenemos que leer.

Por tanto para **buscar información en un fichero 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 es un problema, puesto que no podemos quitar el registro y reordenar el resto. 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, y seguimos copiando hasta el final, el resto de registros al nuevo fichero. De este modo, obtendríamos un nuevo fichero que sería el mismo que teníamos pero sin el registro que queríamos borrar. Por tanto, si se prevé que se va a borrar en el fichero, no es recomendable usar un fichero de este tipo, o sea, secuencial.



Autoevaluación

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 o Falso?

- Verdadero
- Falso

5.6.- Acceso aleatorio.

A veces no necesitamos leer un fichero de principio a fin, sino acceder al fichero como si fuera una base de datos, 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 los registros.

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

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

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

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

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

Para abrirlo en modo lectura y escritura:

```
RandomAccessFile inOut = 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 como **seek** y **skipBytes** para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.

No está basada en el concepto de flujos o streams.

En la siguiente presentación vemos cómo crear un fichero, cómo podemos acceder, y actualizar información en él.



Autoevaluación

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 o Falso?

- Verdadero
- Falso

6.- Aplicaciones del almacenamiento de información en ficheros.

¿Has pensado la **diversidad de ficheros** que existe, según la información que se guarda?

Las fotos que haces con tu cámara digital, o con el móvil, se guardan en ficheros. Así, existe una gran cantidad de ficheros de imagen, según el método que usen para guardar la información. Por ejemplo, tenemos los ficheros de extensión: .jpg, .tiff, gif, .bmp, etc.

La música que oyes en tu mp3 o en el reproductor de mp3 de tu coche, está almacenada en ficheros que almacenan la información en formato mp3.

Los sistemas operativos, como Linux, Windows, etc., están constituidos por un montón de instrucciones e información que se guarda en ficheros.

El propio código fuente de los lenguajes de programación, como Java, C, etc., se guarda en ficheros de texto plano la mayoría de veces.

También se guarda en ficheros las películas en formato .avi, .mp4, etc.

Y por supuesto, se usan mucho actualmente los ficheros XML, que al fin y al cabo son ficheros de texto plano, pero que siguen una estructura determinada. XML se emplea mucho para el intercambio de información estructurada entre diferentes plataformas.

```
<?xml version="1.0"?>
<quiz>
<question>
Who was the forty-second
president of the U.S.A.?
</question>
<answer>
William Jefferson Clinton
</answer>
<!-- Note: We need to add
more questions later.-->
</quiz>
```

XML



Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

Un fichero .bmp guarda información de música codificada. ¿Verdadero o falso?

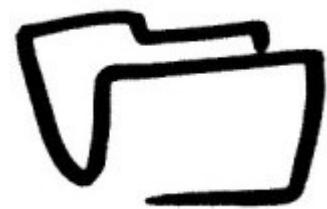
- Verdadero Falso

7.- Utilización de los sistemas de ficheros.

Has visto en los apartados anteriores cómo operar en ficheros: abrirlos, cerrarlos, escribir en ellos, etc.

Lo que no hemos visto es lo relativo a crear y borrar directorios, poder filtrar archivos, es decir, buscar sólo aquellos que tengan determinada característica, por ejemplo, que su extensión sea: .txt.

Ahora veremos cómo hacer estas cosas, y también como borrar ficheros, y crearlos, aunque crearlos ya lo hemos visto en algunos ejemplos anteriores.



7.1.- Clase File.

La clase File proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase **File** representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre dado podría ser que no existiera, por ello, habrá que controlar las posibles excepciones.

Al trabajar con **File**, las rutas pueden ser:

- ✓ Relativas al directorio actual.
- ✓ Absolutas si la ruta que le pasamos como parámetro empieza por
 - ➡ La barra "/" en Unix, Linux.
 - ➡ Letra de unidad (C:, D:, etc.) en Windows.
 - ➡ UNC (universal naming convention) en windows, como por ejemplo:

```
File miFile=new File("\\\\mimaquina\\\\download\\\\prueba.txt");
```

A través del objeto **File**, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto **file**, podemos hacer las siguientes operaciones con él:

- ✓ **Renombrar** el archivo, con el método **renameTo()**. El objeto **File** dejará de referirse al archivo renombrado, ya que el **String** con el nombre del archivo en el objeto **File** no cambia.
- ✓ **Borrar** el archivo, con el método **delete()**. También, con **deleteOnExit()** se borra cuando finaliza la ejecución de la máquina virtual Java.
- ✓ **Crear** un nuevo fichero con un nombre único. El método estático **createTempFile()** crea 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.
- ✓ **Establecer** la fecha y la hora de modificación del archivo con **setLastModified()**. Por ejemplo, se podría hacer: **new File("prueba.txt").setLastModified(new Date().getTime())**; para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso *prueba.txt*.
- ✓ **Crear** un directorio con el método **mkdir()**. También existe **mkdirs()**, que crea los directorios superiores si no existen.
- ✓ **Listar** el contenido de un directorio. Los métodos **list()** y **listFiles()** listan el contenido de un directorio. **list()** devuelve un vector de **String** con los nombres de los archivos, **listFiles()** devuelve un vector de objetos **File**.
- ✓ **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático **listRoots()**.



Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

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

- Verdadero Falso

7.2.- Interface FilenameFilter.

En ocasiones nos interesa ver la lista de los archivos que encajan con un determinado criterio. Así, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que indiquemos, etc.

El interface **FilenameFilter** se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. 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 cuyo nombre se indica en el parámetro **nombre** aparezca en la lista de los ficheros del directorio indicado por el parámetro **dir**.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta **c:\ejerjava** que tengan la extensión **.txt**. Usamos **try** y **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;
    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 fichero=new File("c:\\ejerjava\\");
            String[] listadeArchivos = fichero.list();
            listadeArchivos = fichero.list(new Filtro(".txt"));
            int numarchivos = listadeArchivos.length ;

            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
            else
            {
                for(int conta = 0; conta < listadeArchivos.length; conta++)
                    System.out.println(listadeArchivos[conta]);
            }
        }
        catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```



Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa:

Una clase que implemente **FileNameFilter** puede o no implementar el método **accept**.
¿Verdadero o Falso?

- Verdadero
- Falso

7.3.- Creación y eliminación de ficheros y directorios.

Podemos **crear un fichero** del siguiente modo:

- ✓ Creamos el objeto que encapsula el fichero, por ejemplo, suponiendo que vamos a crear un fichero llamado miFichero.txt, en la carpeta C:\prueba, haríamos:

```
File fichero = new File("c:\\prueba\\miFichero.txt");
```

- ✓ 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:

```
fichero.createNewFile()
```

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

- ✓ Fijamos el nombre de la carpeta y del fichero con:

```
File fichero = new File("C:\\prueba", "agenda.txt");
```

- ✓ Comprobamos si existe el fichero con **exists()** y si es así lo borramos con:

```
fichero.delete();
```

Para **crear directorios**, podríamos hacer:

```
import java.io.File;

public class CrearDIREC {

    public static void main(String[] args) {
        try {
            String directorio = "C:\\\\micarpeta";
            // Crear un directorio
            boolean exito = (new File(directorio)).mkdir();
            if (exito)
                System.out.println("Directorio: " + directorio + " creado");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

Para **borrar un directorio** con Java tenemos que borrar 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 el elemento es un directorio, lo sabemos mediante el método **isDirectory**,

8.- Almacenamiento de objetos en ficheros. Persistencia. Serialización.

¿Qué es la **serialización**? Es un proceso por el que **un objeto se convierte en una secuencia de bytes** con la que más tarde se podrá reconstruir el valor de sus variables. Esto permite guardar un objeto en un archivo.

Para serializar un objeto:

- ✓ éste debe **implementar el interface** `java.io.Serializable`. Este interface no tiene métodos, sólo se usa para informar a la JVM (Java Virtual Machine) que un objeto va a ser serializado.
- ✓ Todos los objetos incluidos en él tienen que implementar el interfaz `Serializable`.

Todos los **tipos primitivos en Java son serializables** por defecto. (Al igual que los arrays y otros muchos tipos estándar).

Para leer y escribir objetos serializables a un stream se utilizan las clases `java.ObjectInputStream` y `ObjectOutputStream`.

En el siguiente ejemplo se puede ver cómo leer un objeto serializado que se guardó antes. En este caso, se trata de un String serializado:

```
FileInputStream fich = new FileInputStream("str.out");
ObjectInputStream os = new ObjectInputStream(fich);
Object o = os.readObject();
```

Así vemos que `readObject` lee un objeto desde el flujo de entrada `fich`. Cuando se leen objetos desde un flujo, se debe tener en cuenta qué tipo de objetos se esperan en el flujo, y se han de leer en el mismo orden en que se guardaron.



Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

Para serializar un fichero basta con implementar el interface `Serializable`. ¿Verdadero o falso?

- Verdadero Falso

8.1.- Serialización: utilidad.

La serialización en Java se desarrolló para utilizarse con RMI (Invocación Remota de Métodos). RMI necesitaba un modo de convertir los parámetros necesarios a enviar a un objeto en una máquina remota, y también para devolver valores desde ella, en forma de flujos de bytes. Para datos primitivos es fácil, pero para objetos más complejos no tanto, y ese mecanismo es precisamente lo que proporciona la serialización.

El método `writeObject` se utiliza para guardar un objeto a través de un flujo de salida. El objeto pasado a `writeObject` debe implementar el interfaz `Serializable`.

```
FileOutputStream fosal = new FileOutputStream("cadenas.out");
ObjectOutputStream oos = new ObjectOutputStream(fosal);
Oos.writeObject();
```

La serialización de objetos se emplea también en la arquitectura de componentes software JavaBean. Las clases bean se cargan en herramientas de construcción de software visual, como NetBeans. Con la paleta de diseño se puede personalizar el bean asignando fuentes, tamaños, texto y otras propiedades.

Una vez que se ha personalizado el bean, para guardarla, se emplea la serialización: se almacena el objeto con el valor de sus campos en un fichero con extensión .ser, que suele emplazarse dentro de un fichero .jar.



Para saber más

En este enlace a puedes ver un vídeo en el que se crea una aplicación sobre serialización.