

**MP0485 Programación
UF8. Gestión de ficheros**

**9.1. Lectura /
Escritura de ficheros**

Índice

☰	Objetivos	3
☰	La clase File	4
☰	Obteniendo información de un chero	10
☰	Obteniendo información de una carpeta	13
☰	Clases que representan flujos de datos	17
☰	Flujos de datos en formato Unicode de 16 bits	20
☰	Flujos de bytes (información binaria)	22
☰	Escrutura en un fichero de texto	24
☰	Lectura de un fichero de texto	31
☰	Escritura en un fichero binario	37
☰	Lectura de un fichero binario	43
☰	Lectura de teclado con InputStreamReader y BufferedReader	47
☰	Gestión de excepciones en la lectura/escritura	49
☰	Introducción a la clase Scanner	51
☰	Lectura de cadenas de texto	52
☰	Lectura desde teclado	57
☰	Lectura de ficheros de texto	61
☰	Resumen	62

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- 1 Obtener información sobre archivos y carpetas haciendo uso de la clase Java *File*.
- 2 Escribir programas Java que realicen operaciones de lectura y escritura de ficheros mediante el uso de la jerarquía de clases manejadoras de flujos de datos.
- 3 Gestionar las excepciones que pueden producirse durante las operaciones de lectura y escritura de ficheros.
- 4 Escribir programas Java utilizando la clase *Scanner* para leer cadenas de texto, introducir datos por teclado o leer archivos de texto.

¡Ánimo y adelante!

La clase File

La clase *File*, situada en el paquete *java.io*, nos permite obtener información sobre archivos y carpetas.

Cada objeto *File* construido vendrá a representar a un determinado archivo o a una determinada carpeta dentro del sistema de archivos.

El constructor de la clase *File* está sobrecargado, de manera que podemos crear un nuevo objeto *File* con cualquiera de estos tres formatos:

1 *File (String path).*

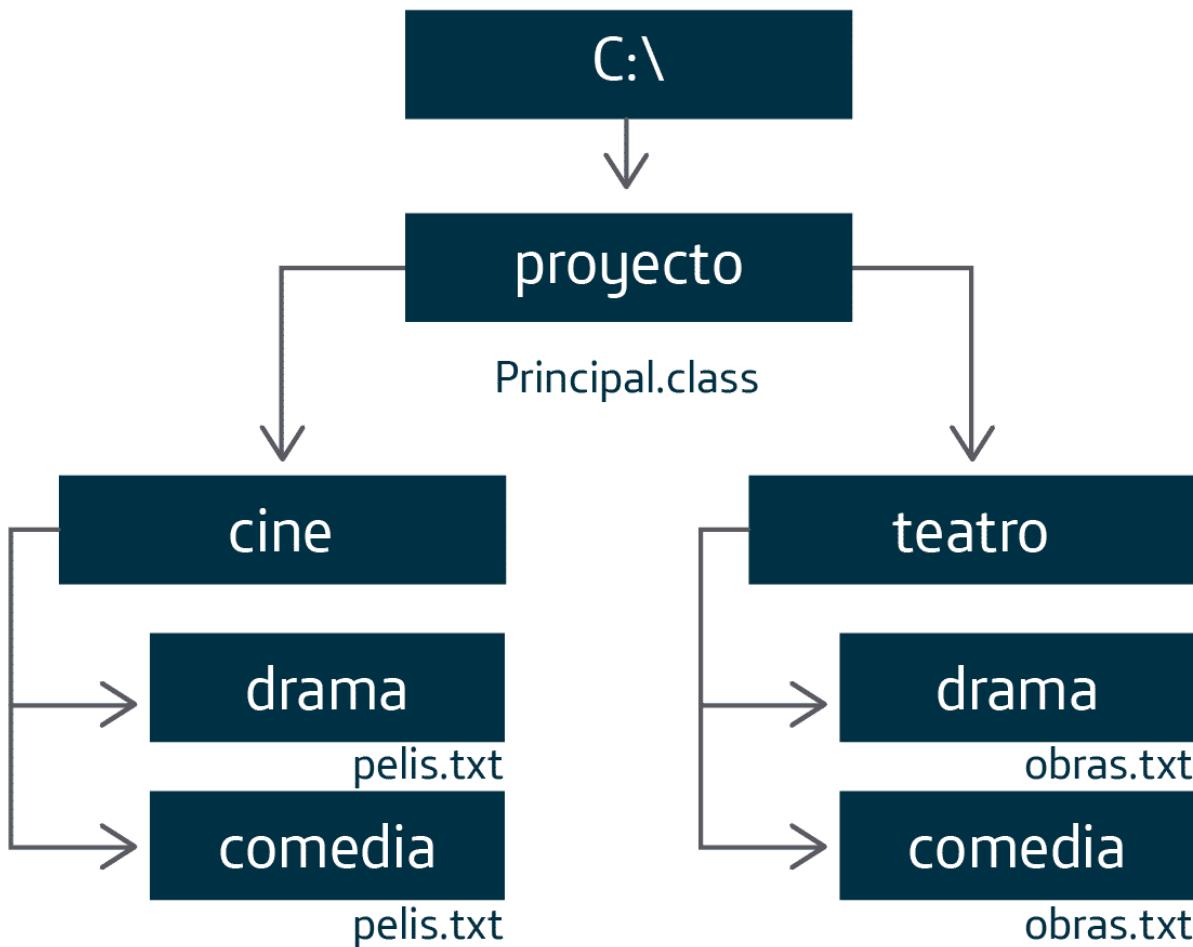
2 *File (String path, String nameFile or path).*

3 *File (File path, String nameFile or path).*

Donde *Path* representa la ruta dentro de la estructura de carpetas y podrá especificarse de forma absoluta o relativa.

- Una ruta absoluta se especifica a partir de la raíz del disco.
- Una ruta relativa se especifica a partir de la ubicación actual, es decir, a partir de la ubicación de la clase Java que intenta acceder a dicha ruta.

Imagina que vas a ejecutar un programa Java situado en C:\proyecto con el nombre Principal.class.



Dentro de la carpeta *proyecto* además existen las carpetas *cine* y *teatro* con las subcarpetas *drama* y *comedia*, tal y como puedes apreciar en la imagen.

Dentro de cada carpeta *drama* y *comedia* además tenemos ficheros de texto.

Utilizaremos esta estructura como ejemplo para mostrar varias formas de crear objetos *File*.

A continuación veremos varios ejemplos de construcción de objetos de la clase *File* utilizando tanto rutas absolutas como relativas.

Usando rutas absolutas

```
File fich = new File("C:/proyecto/cine/drama/pelis.txt");
```

El objeto *fich* representa el fichero especificado en el argumento. Hemos utilizado el primer constructor.

```
File fich = new File("C:/proyecto/cine/drama", "pelis.txt");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String nameFile)*.

```
File carp = new File("C:/proyecto/cine/drama");
```

El objeto *carp* representa a la carpeta *drama*. Hemos utilizado el primer constructor.

```
File carp = new File("C:/proyecto/", "cine/drama");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String subPath)*.

```
File carp = new File("C:/proyecto/cine/drama");
File fich = new File(carp, "pelis.txt");
```

En este ejemplo estamos creando el objeto *fich*, que representa un fichero a partir del objeto *carp*, que representa la carpeta donde está ubicado el fichero. Estamos utilizando el tercer constructor para crear el objeto *fich*: *File (File path, String nameFile)*.

i CURIOSIDAD: usamos el símbolo / en lugar del símbolo \ dentro de la ruta porque el símbolo \ es un carácter de escape especial para Java y nos ocasionaría error de compilación. Recuerda que en varias ocasiones has incluido la combinación "\n" en una cadena de texto para generar un retorno de carro, esto es porque el símbolo \ va seguido de alguno de los caracteres especiales que tienen un significado específico (un retorno de carro en el caso de la n).

Podemos especificar la ruta de dos formas distintas:

C:/proyecto/cine/drama
o
C:\\\\proyecto\\\\cine\\\\drama

En el segundo caso hemos colocado el símbolo \ como un carácter especial de escape y así no tenemos errores de compilación.

[Obtén información sobre los caracteres especiales de escape Java.](#)

Usando rutas relativas

Ahora vamos a presentar exactamente los mismos ejemplos anteriores pero con rutas relativas, es decir, a partir de la ubicación del programa, que en el ejemplo de la imagen es la carpeta *C:\Proyecto*. En este caso usaremos el carácter de escape \ para que te acostumbres a los dos sistemas, aunque podrías hacerlo con el carácter / igualmente.

```
File fich = new File("cine\\drama\\pelis.txt");
```

El objeto *fich* representa el fichero especificado en el argumento.

```
File fich = new File("cine\\drama", "pelis.txt");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String nameFile)*.

```
File carp = new File("cine\\drama");
```

El objeto *carp* representa a la carpeta *drama*. Hemos utilizado el primer constructor.

```
File carp = new File("cine\\drama");
File fich = new File(carp, "pelis.txt");
```

En este ejemplo estamos creando el objeto *fich*, que representa un fichero a partir del objeto *carp*, que representa la carpeta donde está ubicado el fichero. Estamos utilizando el tercer constructor para crear el objeto *fich*: *File (File path, String nameFile)*.

¿Y qué podemos hacer con un objeto *File*?

La clase *File* nos permite las siguientes operaciones:

- 1 **Obtener información sobre archivos:** número de bytes que ocupa, propiedades del archivo (si es de solo lectura, oculto, etc.), ruta donde se encuentra, etc.
- 2 **Obtener información sobre carpetas:** propiedades de la carpeta, archivos y subcarpetas que contiene, etc.
- 3 **Borrar archivos y carpetas.**
- 4 **Crear archivos y carpetas.**

Aunque la clase *File* permita borrar y crear archivos y carpetas, no permite operaciones de lectura y escritura.

¿Qué es lo que **NO permite** la clase *File*?

- 1 **No permite leer** el contenido de un fichero. Para eso están los flujos de datos de lectura que estudiaremos en otro apartado de esta misma unidad.
- 2 **No permite escribir** dentro de un fichero. Para eso están los flujos de datos de escritura que estudiaremos en otro apartado de esta misma unidad.

Obteniendo información de un fichero

Ya tengo mi objeto *File*.

¿Ahora qué hago con él?

Prueba a ejecutar este pequeño programa. Crea un objeto *File* que representa a un archivo llamado *pelis.txt* y después obtiene la siguiente información: número de bytes que ocupa, nombre del archivo, ruta, propiedades del fichero (si es oculto, si se puede escribir en él, si se puede leer).

```
import java.io.File;

public class Principal {
    public static void main(String args[]) {
        File fich = new File("C:/proyecto/cine/drama/pelis.txt");
        if (fich.exists()) {
            System.out.println("Existe el fichero");
            System.out.println("Nº de bytes que ocupa: " + fich.length());
            System.out.println("Nombre de archivo: " + fich.getName());
            System.out.println("Ruta: " + fich.getPath());
            System.out.println("¿Es un fichero oculto? " + fich.isHidden());
            System.out.println("¿Está permitida la escritura? " + fich.canWrite());
            System.out.println("¿Está permitida la lectura? " + fich.canRead());
        }
        else {
            System.out.println("El fichero no existe");
        }
    }
}
```

La clase *File* también nos permite crear o eliminar un fichero.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File fich = new File("C:/proyecto/cine/drama/pelisdeterror.txt");
        boolean ok = fich.createNewFile();
        if (ok)
            System.out.println("El fichero se ha creado con éxito");
        else
            System.out.println("El fichero no ha podido crearse");
    }
}
```

Como has podido comprobar por el ejemplo anterior, podemos instanciar un objeto *File* que represente un fichero que no existe y después crearlo con el método *createNewFile()*, que creará físicamente el fichero con el nombre y ruta especificada en el constructor de *File*. El método *createNewFile()* devuelve *true* si el fichero se ha creado y *false* de lo contrario.

Si pruebas a ejecutar el programa dos veces la primera mostrará el mensaje "El fichero se ha creado con éxito", siempre y cuando la ruta especificada exista. La segunda vez mostrará el mensaje "El fichero no ha podido crearse", ya que no se puede crear un fichero que ya existe.

**Prueba a ejecutar una vez más el programa, pero ahora con una ruta que no exista.
Comprueba que se desencadena una excepción en tiempo de ejecución.**

El método *createNewFile()* puede provocar excepciones de tipo **IOException**; debemos encerrarlo en un *try ... catch* o bien propagar la excepción con un *throws*.

Ten en cuenta que la operación de crear un fichero en disco puede tener varias situaciones de excepción:

- La ruta especificada no existe.
- Intentamos crear un fichero en una carpeta que está protegida contra escritura.
- El disco utilizado está deteriorado o lleno.

Ahora vamos a eliminar el fichero que acabamos de crear

```
import java.io.File;

public class Principal {
    public static void main(String args[]) {
        File fich = new File("C:/proyecto/cine/drama/pelisdeterror.txt");
        boolean ok = fich.delete();
        if (ok)
            System.out.println("El fichero se ha borrado con éxito");
        else
            System.out.println("El fichero no ha podido borrarse");
    }
}
```

El método *delete()* elimina el fichero y devuelve *true* si la operación se ha completado con éxito, de lo contrario devuelve *false*.

Obteniendo información de una carpeta

En esta ocasión vamos a trabajar con un objeto *File* que representa una carpeta.

Pega este código en un proyecto Eclipse y ejecútalo:

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\\\proyecto\\\\cine\\\\drama");
        if (carp.exists()) {
            System.out.println("Existe la carpeta");
            System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
            String[] contenido = carp.list();
            System.out.println("Archivos o carpetas que contiene: " + contenido.length);
            for (String nombre : contenido) {
                System.out.println(nombre);
            }
        } else
            System.out.println("No existe la carpeta");
    }
}
```

Vamos a analizar el programa poco a poco:

```
System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
```

El método *canWrite()* devuelve *true* si la carpeta tiene permisos de escritura, es decir, no está protegida como "solo lectura".

```
String[] contenido = carp.list();
System.out.println("Archivos o carpetas que contiene: " + contenido.length)
```

El método *list()* devuelve un *array* de objetos *String* con los nombre de los archivos o subcarpetas contenidos en la carpeta que representa el objeto *carp*. El *array* devuelto lo estamos guardando en la variable *contenido*, que será un *array*. La propiedad *length* del *array* contiene el número de elementos, que en este caso coincide con el número de archivos o carpetas.

```
for (String nombre : contenido) {
    System.out.println(nombre);
}
```

Por último, estamos utilizando una estructura *for each* para recorrer los elementos del *array* y así mostrar en pantalla los nombres de los archivos y carpetas.

Vamos a dar otro paso más

Si podemos obtener un *array* con los nombres de archivos y carpetas, también podremos utilizar estos nombres para construir nuevos objetos *File* para obtener más información sobre cada uno de ellos. ¡Vamos a comprobarlo! Accederemos a una carpeta que tenga más contenido, por ejemplo la carpeta *Windows*, que casi seguro está situada en *C:\Windows*.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\windows");
        if (carp.exists()) {
            System.out.println("Existe la carpeta");
            System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
            String[] contenido = carp.list();
            System.out.println("Archivos o carpetas que contiene: " + contenido.length());
            for (String nombre : contenido) {

                File f = new File(carp.getPath(), nombre);
                if (f.isDirectory()) {
                    System.out.println(nombre + ", " + " carpeta");
                }
                else {
                    System.out.println(nombre + ", " + " fichero, " + f.length() + " bytes");
                }
            }
        }
        else
            System.out.println("No existe la carpeta");
    }
}
```

Este ejemplo es muy parecido al anterior, pero ahora estamos accediendo a la carpeta *Windows*, comprobando por cada elemento si se trata de una carpeta o archivo. En caso de ser un archivo muestra el número de bytes que ocupa.

La clase *File* también sirve para crear y eliminar carpetas.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\\\prueba");
        boolean ok = carp.mkdir();
        if (ok)
            System.out.println("La carpeta se ha creado con éxito");
        else
            System.out.println("La carpeta no ha podido crearse");
    }
}
```

El método *mkdir()* crea la carpeta representada por el objeto *File* si no existe. Devuelve *true* si la carpeta se ha podido crear con éxito y *false* de lo contrario.

Ahora vamos a borrar la carpeta que acabamos de crear:

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

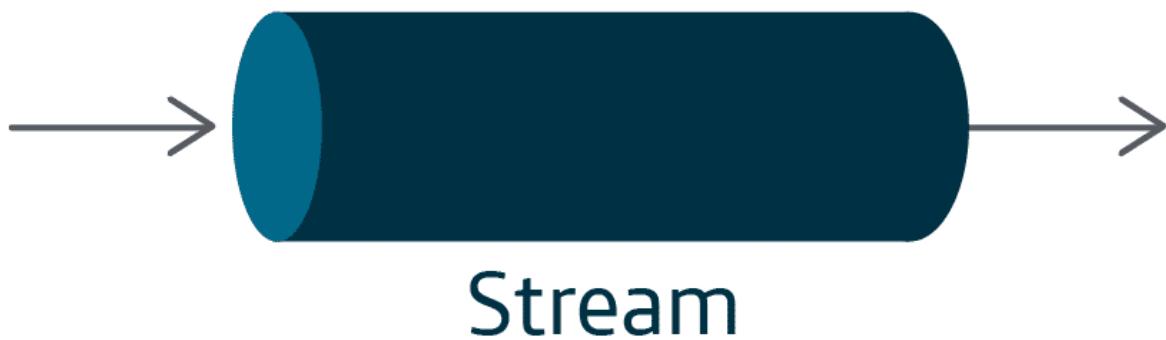
public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\\\prueba");
        boolean ok = carp.delete();
        if (ok)
            System.out.println("La carpeta se ha borrado con éxito");
        else
            System.out.println("La carpeta no ha podido borrarse");
    }
}
```

El método *delete()* elimina la carpeta. Devuelve *true* si la carpeta se ha eliminado con éxito y *false* de lo contrario. El método *delete()* no permite eliminar una carpeta que tenga algo dentro, es decir, no puede contener ningún archivo ni subcarpeta.

Clases que representan flujos de datos

Toda la información que se transmite a través de un ordenador fluye desde una entrada hacia una salida. Para transmitir información, Java utiliza unos objetos especiales denominados *streams* (flujos o corrientes).

Toda operación de lectura o escritura de ficheros requiere del uso de un flujo de datos o *stream*.



Flujo de datos o *stream*.

Los *stream* permiten transmitir secuencias ordenadas de datos desde un origen a un destino. El origen y el destino puede ser un fichero, un *String* o un dispositivo (lectura de teclado, escritura en pantalla).

Java dispone de dos grupos de flujos de datos:

1

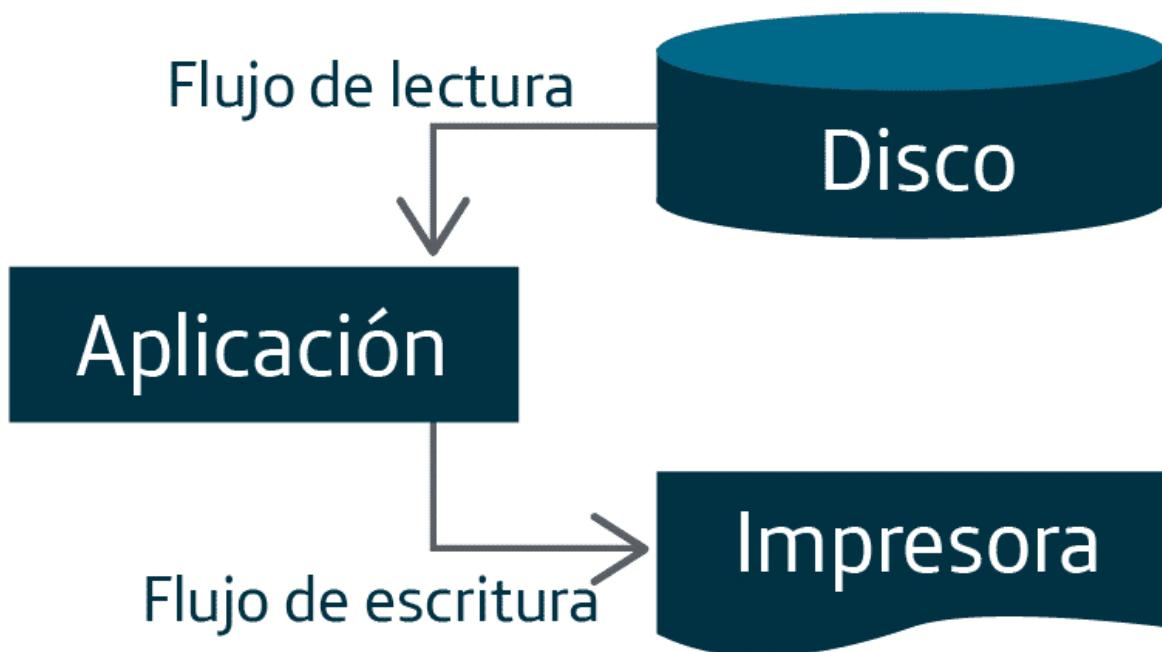
Flujos de entrada o lectura (*input streams*): los datos fluyen desde el fichero o dispositivo hacia el programa.

2

Flujos de salida o escritura (*output streams*): los datos fluyen desde el programa hacia el fichero o dispositivo.

Java no dispone de flujos de entrada / salida

Java no dispone de clases que representen flujos de lectura y escritura. Si necesitamos leer y escribir de un fichero necesitamos dos flujos distintos: un flujo de entrada o lectura y otro de salida o escritura.



Organigrama de lectura y escritura de archivos.

Todo proceso de lectura o escritura de datos consta de tres pasos:

1

Abrir el flujo de datos de lectura o de escritura.

2

Leer o escribir datos a través del flujo abierto.

3

Cerrar el flujo de datos.

Las clases manejadoras de flujos de datos

Todas las clases que representan flujos de datos están ubicadas en el paquete *java.io*.

Dentro del paquete *java.io* disponemos de varias clases para representar flujos de datos. Están organizadas en dos grandes grupos:

1

Flujos de datos en formato Unicode de 16 bits: derivados de las clases abstractas *Reader* y *Writer*.

2

Flujos de bytes (información binaria): derivados de las clases abstractas *InputStream* y *OutputStream*.



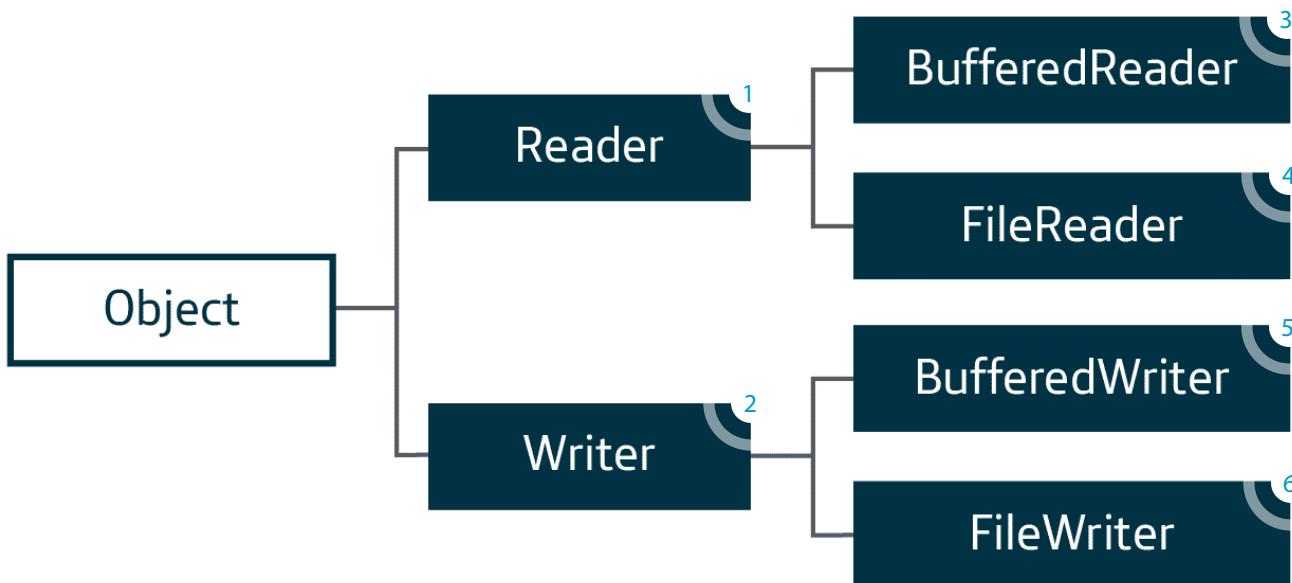
En los dos siguientes apartados estudiaremos más detenidamente las clases más importantes del paquete *java.io*.

Flujos de datos en formato Unicode de 16 bits

Todas las clases que representan flujos de datos (*streams*) en formato Unicode de 16 bits derivan de las clases abstractas *Reader* y *Writer*. En la imagen hemos reflejado las clases más importantes dentro de esta categoría.

Los flujos de datos, además de diferenciarse según sean de entrada o salida, también se distinguen por su cercanía al dispositivo. En este sentido hay dos tipos de flujos de datos:

- **Iniciadores:** vuelcan o recogen datos directamente del dispositivo.
- **Filtros:** se sitúan entre el *stream* iniciador y el programa.



1 Reader

Clase abstracta de la que derivan todas las clases que representan flujos de entrada de caracteres Unicode de 16 bits.

2 Writer

Clase abstracta de la que derivan todas las clases que representan flujos de salida de caracteres Unicode de 16 bits.

3 **BufferedReader**

Permite mejorar la velocidad de transmisión en la lectura de un fichero proporcionando un *buffer*. Entra dentro de la categoría de filtro y trabaja en colaboración con un objeto *FileReader*.

4 **FileReader**

Permite leer caracteres de un fichero. Es iniciador y puede trabajar en conjunto con la clase *BufferedReader*, que actúa como filtro y mejora la eficiencia de la lectura.

5 **BufferedWriter**

Permite mejorar la velocidad de escritura en un fichero proporcionando un *buffer*. Entra dentro de la categoría de filtro y trabaja en colaboración con la clase *FileWriter*.

6 **FileWriter**

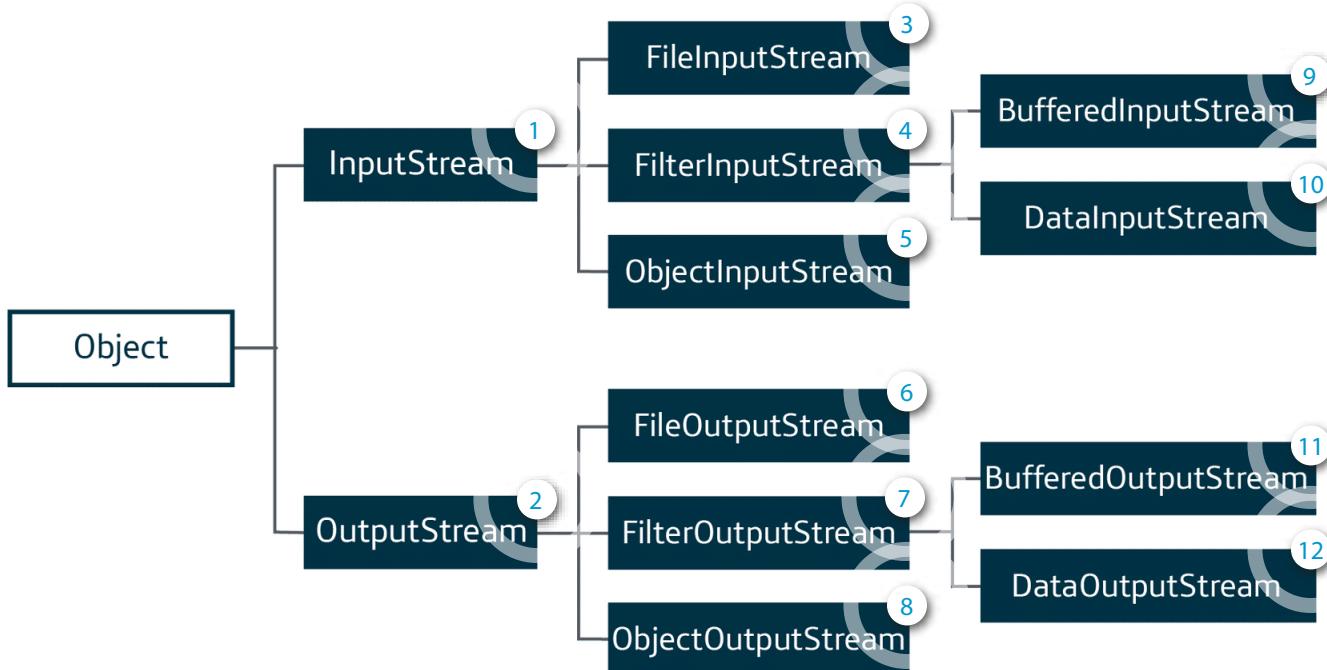
Permite escribir caracteres en un fichero. Es iniciador y puede trabajar en conjunto con la clase *BufferedWriter*, que actúa como filtro y mejora la eficiencia de la escritura.

Flujos de bytes (información binaria)

Todas las clases que representan flujos de datos (*streams*) en formato binario derivan de las clases abstractas *InputStream* y *OutputStream*. En la imagen hemos reflejado las clases más importantes dentro de esta categoría.

Igual que ocurría con los flujos de caracteres Unicode, los flujos de bytes también se subdividen en:

- **Iniciadores:** vuelcan o recogen datos directamente del dispositivo.
- **Filtros:** se sitúan entre el *stream* iniciador y el programa.



1 **InputStream**

Clase abstracta de la que derivan todas las clases que representan flujos de entrada de bytes.

2 **OutputStream**

Clase abstracta de la que derivan todas las clases que representan flujos de salida de bytes.

3 FileInputStream

Permite leer bytes de un fichero. Actúa como iniciador.

4 FilterInputStream

Clase base de la que derivan las siguientes subclases que actúan como filtro, mejorando las operaciones de lectura: *BufferedInputStream* y *DataInputStream*.

5 ObjectInputStream

Permite la lectura de objetos guardados en disco. Actúa como filtro y requiere un iniciador, por ejemplo un *FileInputStream*.

6 FileOutputStream

Permite escribir bytes en un fichero. Actúa como iniciador.

7 FilterOutputStream

Clase base de la que derivan las siguientes subclases que actúan como filtro, mejorando las operaciones de escritura: *BufferedOutputStream* y *DataOutputStream*.

8 ObjectOutputStream

Permite la escritura de objetos en disco. Actúa como filtro y requiere un iniciador, por ejemplo un *FileOutputStream*.

9 BufferedInputStream

Permite mejorar la eficiencia de la lectura de un fichero proporcionando un *buffer*. Trabaja en colaboración con un objeto iniciador, por ejemplo un *FileInputStream*.

10 DataInputStream

DataInputStream permite leer datos de un fichero recogiéndolos directamente como tipos de datos primitivos (*int*, *float*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como *FileInputStream*.

11 BufferedOutputStream

Permite mejorar la eficiencia de la escritura en un fichero proporcionando un *buffer*. Trabaja en colaboración con un objeto iniciador, por ejemplo un *FileOutputStream*.

12 DataOutputStream

Permite escribir datos en un fichero directamente como tipos de datos primitivos (*int*, *float*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como *FileOutputStream*.

Escritura en un fichero de texto

En este apartado tendrás oportunidad de practicar con los flujos de datos para la escritura en ficheros de texto, principalmente en formato Unicode de 16 bits.

Generalmente las operaciones de lectura / escritura requieren de un objeto iniciador que se comunique directamente con el dispositivo y un filtro con el que realizar la lectura / escritura eficientemente.

Escritura en formato Unicode con *FileWriter* y *BufferedWriter*

En el siguiente programa utilizaremos las clases *FileWriter* (iniciador) y *BufferedWriter* (filtro) para escribir el título de tres películas en un fichero de texto. Para ello realizaremos los tres pasos típicos asociados a cualquier tarea de escritura en ficheros:

- 1 Abrir fichero para escritura.
- 2 Escribir líneas en el fichero.
- 3 Cerrar el fichero.

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

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileWriter file;
        try {
            file = new FileWriter("C:\\\\cine\\\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedWriter buffer = new BufferedWriter(file);
        try {
            buffer.write("¡Bienvenido, Mister Marshall!");
            buffer.newLine();
            buffer.write("Con la muerte en los talones");
            buffer.newLine();
            buffer.write("Muerte de un ciclista");
            buffer.newLine();
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el buffer y el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Ahora vamos a analizar las partes más importantes del programa:

```
file = new FileWriter("C:\\cine\\peliculas.txt");
```

```
file = new FileWriter("C:\\cine\\peliculas.txt");
```

Al construir un objeto de la clase *FileWriter* estamos abriendo el fichero *peliculas.txt*, dejándolo preparado para escritura.

```
BufferedWriter buffer = new BufferedWriter(file);
```

```
BufferedWriter buffer = new BufferedWriter(file);
```

Las operaciones de escritura las realizaremos a través del objeto *buffer* de la clase *BufferedWriter*, que nos proporciona métodos para la escritura eficiente. El constructor de la clase *BufferedWriter* requiere un objeto *FileWriter*.

```
buffer.write("¡Bienvenido, Mister Marshall!");
```

```
buffer.write("¡Bienvenido, Mister Marshall!");
```

El método *write(String texto)* de la clase *BufferedWriter* escribe texto en el fichero.

```
buffer.newLine();
```

```
buffer.newLine();
```

El método *newLine()* de la clase *BufferedWriter* escribe un retorno de carro en el fichero.

```
buffer.close(); y file.close();
```

```
buffer.close();
```

```
file.close();
```

Por último tenemos que cerrar los objetos *BufferedWriter* y *FileWriter*.



Habráis observado que, aunque el fichero **peliculas.txt** no existía previamente, el constructor de la clase **FileWriter** lo ha creado.

La sentencia:

```
file = new FileWriter("C:\\cine\\peliculas.txt");
```

no solo nos ha permitido abrir el fichero para escritura, sino que también lo ha creado físicamente. Si pruebas a ejecutar varias veces el programa verás que no se van añadiendo nuevas líneas, siempre hay tres. Cada vez que ejecutamos vuelve a crear el fichero sobrescribiendo el anterior. **Si lo que deseamos es añadir líneas a un fichero existente solo tenemos que pasar un argumento más al constructor de *FileWriter* con el valor *true*.**

```
file = new FileWriter("C:\\cine\\peliculas.txt", true);
```

Así abrimos el fichero para añadir nuevas líneas. Si el fichero no existe, lo crea, pero si existe lo abre para añadir.

El método *write* de la clase *BufferedWriter* también puede recibir un número entero, que escribirá en el fichero el carácter asociado en Unicode con dicho número.

```
buffer.write(65);
```

Esta sentencia escribiría en el fichero una 'A', carácter asociado al valor numérico 65.

Pon a prueba este ejemplo, que escribe en un fichero los caracteres Unicode asociados a los números 0 a 255.

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

public class Principal {
    public static void main(String args[])  {

        // Abrir fichero para escritura
        FileWriter file;
        try {
            file = new FileWriter("C:\\\\cine\\\\caracteres.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedWriter buffer = new BufferedWriter(file);
        try {
            for (int i=0; i<=255; i++) {
                buffer.write(i+": "); // Escritura de un St
                buffer.write(i); // Escritura del carácter
                buffer.newLine();
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el buffer y el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }
}
```

También es posible escribir en ficheros de texto con un flujo de datos en formato binario, pero en ese caso hay que escribir uno a uno cada byte que corresponda al texto.

Pon a prueba el siguiente ejemplo:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileOutputStream file;
        try {
            file = new FileOutputStream("C:\\\\cine\\\\peliculas2.txt");
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }
        String texto = "Con la muerte en los talones";

        // Escribir el texto en el fichero carácter a carácter.
        try {
            for (int i=0; i<texto.length(); i++) {
                file.write(texto.charAt(i));
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }
}
```

Observa que hemos realizado todo el proceso de escritura exclusivamente con un objeto iniciador (*FileOutputStream*). También podemos hacer que este objeto iniciador actúe en colaboración con un filtro para mejorar el rendimiento proporcionando un *buffer*. El siguiente programa es igual que el anterior pero incorpora un *buffer*.

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileOutputStream file;
        BufferedOutputStream buffer;
        try {
            file = new FileOutputStream("C:\\\\cine\\\\peliculas2.t");
            buffer = new BufferedOutputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }
        String texto = "Con la muerte en los talones";

        // Escribir el texto en el fichero carácter a carácter.
        try {
            for (int i=0; i<texto.length(); i++) {
                buffer.write(texto.charAt(i));
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }
}
```

Lectura de un fichero de texto

En este apartado tendrás oportunidad de practicar con los flujos de datos para la lectura de ficheros de texto, principalmente en formato Unicode de 16 bits.

Leer el fichero *peliculas.txt* utilizando flujos de datos en formato UNICODE de 16 bits

Comenzaremos por utilizar las clases *FileReader* (iniciador) y *BufferedReader* (filtro) para realizar la lectura del fichero *peliculas.txt* creado en el apartado anterior.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para lectura
        FileReader file;
        try {
            file = new FileReader("C:\\\\cine\\\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedReader buffer = new BufferedReader(file);
        String linea="";
        try {
            linea = buffer.readLine();
            while (linea!=null) {
                System.out.println(linea);
                linea = buffer.readLine();
            }
        } catch (IOException e) {
            System.out.println("Error al leer el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el buffer y el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Analicemos ahora las partes más importantes del programa:

- `file = new FileReader("C:\\cine\\peliculas.txt");`
Con esta sentencia estamos abriendo el fichero *peliculas.txt* para lectura, desencadenando una excepción del tipo *FileNotFoundException* si el fichero no existe o cualquier otra excepción de tipo *IOException* si se produce otro tipo de error que no permita la apertura del fichero.
- `linea = buffer.readLine();`
Con esta sentencia estamos leyendo la siguiente línea del fichero de manera secuencial y guardándola en la variable *linea*. Cuando ya no hay más líneas para leer devuelve *null*; esa es la razón por la que necesitamos una estructura *while* con la condición *linea!=null*.
- `buffer.close();
file.close();`
Finalmente debemos desbloquear el fichero cerrando los flujos de datos de filtro e iniciador.

Leer el ficher *peliculas.txt* utilizando un flujo de datos en formato binario

También podemos leer el contenido de un fichero de texto con un objeto de la clase *FileInputStream*, pero como se trata de un flujo de datos en formato binario, hay que leer uno a uno los bytes correspondientes a cada uno de los caracteres.

Ponlo en práctica ejecutando este programa:

```
import java.io.FileInputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileInputStream file;
        try {
            file = new FileInputStream("C:\\cine\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Leer contenido del fichero carácter a carácter
        int caracter;
        try {
            caracter = file.read(); // Lee un byte y devuelve -1
            while (caracter!=-1) {
                System.out.print((char)caracter);
                caracter = file.read();
            }
        } catch (IOException e) {
            System.out.println("Error al leer el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Analicemos las partes más importantes del programa:

- `file = new FileInputStream("C:\\cine\\peliculas.txt");`
Con esta línea abrimos el fichero para lectura.
- `caracter = file.read();`
Esta sentencia lee el próximo carácter en orden secuencial y devuelve un número entero con el código Unicode asociado al carácter. Cuando llega al final de fichero devuelve -1, esa es la razón por la que usamos una estructura *while* cuya condición es que la variable *caracter* sea distinta de -1. A la hora de mostrar el carácter en pantalla hacemos conversión a tipo *char* para que se vayan mostrando los caracteres y no los códigos.
- `file.close();`
Con esta sentencia cerramos el fichero.

Observa que en esta ocasión hemos realizado la lectura utilizando exclusivamente un objeto de tipo iniciador (*FileInputStream*). También es posible utilizar un objeto que actúe de filtro proporcionando un método más eficiente de lectura. El siguiente programa funciona exactamente igual que el anterior, pero incorpora un filtro.

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

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileInputStream file;
        BufferedInputStream buffer;
        try {
            file = new FileInputStream("C:\\cine\\peliculas.txt");
            buffer = new BufferedInputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Leer contenido del fichero carácter a carácter
        int caracter;
        try {

            caracter = buffer.read(); // Lee un byte y devuelve
            while (caracter!= -1) {
                System.out.print((char)caracter);
                caracter = buffer.read();
            }
        } catch (IOException e) {
            System.out.println("Error al leer el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }
}
```

Escritura en un fichero binario

Los ficheros binarios están formados por secuencias de bytes, pueden contener datos de tipo elemental (*int*, *float*, *double*, etc.) u objetos.

El formato binario de datos es muy útil cuando se quieren representar datos en forma de registros y campos, como si de una base de datos se tratara. En esta ocasión, como ejemplo, vamos a crear un programa que permita añadir los datos de los artículos disponibles en un almacén. Comenzaremos por revisar los pasos que tenemos que seguir para lograrlo:

- 1 Crear una clase llamada *Producto*, que servirá para representar la estructura de cada uno de los artículos del almacén y utilizarla para crear todos los objetos *Producto* que sean necesarios.
- 2 Crearemos un objeto de la clase *FileOutputStream*, que servirá como flujo iniciador para abrir el fichero *almacen.dat*.
- 3 Crearemos un objeto de la clase *DataOutputStream*, que servirá como filtro proporcionando un *buffer* de escritura y lo vincularemos al objeto *FileOutputStream*.
- 4 Escribiremos registros en el *buffer* proporcionado por el objeto *DataOutputStream*.
- 5 Cerraremos los flujos *DataOutputStream* y *FileOutputStream*.

Crea un proyecto Eclipse con el nombre que deseas y crea la clases *Producto* y *Principal*, cuyo código puedes copiar y pegar desde aquí:

```
public class Producto {  
    private String nombre;  
    private float precio;  
    private float unidadesEnExistencia;  
  
    public Producto(String nombre, float precio, float unidadesEnExiste  
        this.nombre = nombre;  
        this.precio = precio;  
        this.unidadesEnExistencia = unidadesEnExistencia;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
    public float getPrecio() {  
        return precio;  
    }  
  
    public float getUnidadesEnExistencia() {  
        return unidadesEnExistencia;  
    }  
  
    @Override  
    public String toString() {  
        return nombre + " Stock: " + this.unidadesEnExistencia + "  
    }  
}
```

Clase que sirve para representar la estructura de cada artículo del almacén.

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Principal {

    public static void main(String[] args) {

        // Creación de 3 objetos producto
        Producto p1 = new Producto("Manzanas Royal Gala",2.50f,7f);
        Producto p2 = new Producto("Dátiles de la tía Julita",3.25f,2);
        Producto p3 = new Producto("Mandarinas Clementinas",2.20f,2);

        FileOutputStream fichero;
        DataOutputStream escritor;

        // Apertura del fichero almacen.dat
        try {
            fichero = new FileOutputStream("almacen.dat", true);
            escritor = new DataOutputStream (fichero);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Escribir datos en el fichero almacen.dat
        try {
            escritor.writeUTF(p1.getNombre());
            escritor.writeFloat(p1.getPrecio());
            escritor.writeFloat(p1.getUnidadesEnExistencia());

            escritor.writeUTF(p2.getNombre());
            escritor.writeFloat(p2.getPrecio());
            escritor.writeFloat(p2.getUnidadesEnExistencia());

            escritor.writeUTF(p3.getNombre());
            escritor.writeFloat(p3.getPrecio());
            escritor.writeFloat(p3.getUnidadesEnExistencia());

        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al escribir");
            System.out.println(e.getMessage());
        }

        try {
            escritor.close();
            fichero.close();
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al cerrar");
            System.out.println(e.getMessage());
        }
    }
}
```

Guardar tres artículos en el fichero *almacen.dat*.

Vamos a repasar las acciones que hemos llevado a cabo:

1

Hemos creado tres objetos de la clase *Producto* con los datos que posteriormente guardaremos en el fichero *almacen.dat*

```
// Creación de 3 objetos producto  
Producto p1 = new Producto("Manzanas Royal Gala",2.50f,7f);  
Producto p2 = new Producto("Dátiles de la tía Julita",3.25f,12f);  
Producto p3 = new Producto("Mandarinas Clementinas",2.20f,25f);
```

2

Hemos creado un objeto de la clase *FileOutputStream*

```
fichero = new FileOutputStream("almacen.dat", true);
```

La construcción del objeto *FileOutputStream*, que actúa como flujo de datos iniciador, nos ha dejado el archivo *almacen.dat* abierto para escritura. Con el segundo argumento asignado a *true* logramos que, si el fichero ya existe, agregue los nuevos datos sin sobrescribir lo anterior.

Como no hemos especificado ninguna ruta para el fichero *almacen.dat*, se creará en la misma carpeta donde se encuentra el proyecto Eclipse.

3

Hemos creado un objeto *DataOutputStream*.

```
escritor = new DataOutputStream (fichero);
```

Construimos el objeto *DataOutputStream* (filtro) pasando como argumento el nuevo objeto *FileOutputStream* para proporcionar un sistema eficiente que permite guardar en disco datos de tipo elemental. En nuestro caso guardaremos un texto en formato UTF y dos valores tipo *float*.

4

Escribimos registros con ayuda del objeto *DataOutputStream* (*buffer*)

```
escritor.writeUTF(p1.getNombre());
escritor.writeFloat(p1.getPrecio());
escritor.writeFloat(p1.getUnidadesEnExistencia());

escritor.writeUTF(p2.getNombre());
escritor.writeFloat(p2.getPrecio());
escritor.writeFloat(p2.getUnidadesEnExistencia());

escritor.writeUTF(p3.getNombre());
escritor.writeFloat(p3.getPrecio());
escritor.writeFloat(p3.getUnidadesEnExistencia());
```

Recuerda que la clase *DataOutputStream* proporciona un *buffer* que permite escribir datos de tipo primitivo en un fichero (*int*, *float*, *double*, etc.) para lo cual utilizamos los siguientes métodos:

- ***writeBoolean(boolean valor)***: escribe un valor de tipo *boolean* en el fichero.
- ***writeByte(byte valor)***: escribe un valor de tipo *byte* en el fichero.
- ***writeBytes(String cadena)***: escribe cada uno de los bytes que forman parte de la cadena especificada en el argumento.
- ***writeChar(int valor)***: escribe el carácter asociado al código pasado como argumento.

- **`writeChars(String cadena)`**: escribe cada uno de los caracteres de la cadena.
- **`writeDouble(double valor)`**: escribe un valor de tipo *double*.
- **`writeFloat(float valor)`**: escribe un valor de tipo *float*.
- **`writeInt(int valor)`**: escribe un valor de tipo *int*.
- **`writeLong(long valor)`**: escribe un valor de tipo *long*.
- **`writeShort(int valor)`**: escribe el valor del argumento y lo almacena como un *short*.
- **`writeUTF(String cadena)`**: escribe la cadena en formato *UTF*.

5

Cerramos los dos flujos de datos (filtro e iniciador) cerrando de esta forma el fichero.

```
escritor.close();
fichero.close();
```

Si ya has ejecutado el programa, comprueba que existe el fichero *almacen.dat* en la carpeta del proyecto. Si lo abres con el bloc de notas comprobarás que no se trata de un archivo de texto, no se puede interpretar a simple vista, se trata de un archivo binario y necesitamos de un programa específico para leerlo.

Lectura de un fichero binario

En este apartado abriremos el fichero *almacen.dat* para lectura y mostraremos en pantalla los registros que contiene, es decir, los artículos del almacén.

Realizaremos los siguientes pasos:

- 1 Crear un objeto de la clase *FileInputStream* dejando abierto el fichero *almacen.dat* para lectura.
- 2 Crear un objeto de la clase *DataInputStream* (filtro) asociado al objeto *FileInputStream* (iniciador) para obtener un *buffer* que permita optimizar la lectura, proporcionando métodos que permitan leer datos elementales de tipo *int*, *float*, *double*, etc.
- 3 Leer datos secuencialmente hasta llegar al final del fichero.
- 4 Cerrar los objetos *DataInputStream* y *FileInputStream*, dejando así cerrado el fichero.

Puedes crear otro proyecto Eclipse para realizar la lectura, pero recuerda que necesitarás copiar la clase *Producto*.

Para leer datos de un fichero binario tienes que saber previamente la estructura que tiene dicho fichero. En nuestro caso no hay duda, sabemos que hemos guardado registros con la estructura *String*, *float*, *float* y justo así es como iremos recuperándolos.

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;

public class Principal {

    public static void main(String[] args) {
        FileInputStream fichero;
        DataInputStream lector;
        try {
            fichero = new FileInputStream("almacen.dat");
            lector = new DataInputStream (fichero);
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        boolean eof = false;
        while (!eof) {
            try {
                String pro = lector.readUTF();
                float pre = lector.readFloat();
                float uni = lector.readFloat();
                Producto p = new Producto(pro, pre, uni);
                System.out.println(p);
            } catch (EOFException e1) {
                eof = true;
            } catch (IOException e2) {
                System.out.println("Ha ocurrido un error al leer el fichero");
                System.out.println(e2.getMessage());
                break; // sale del bucle while
            }
        }

        try {
            lector.close();
            fichero.close();
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Si observas el programa anterior comprobarás que tiene la típica estructura a la que ya te has acostumbrado: apertura del fichero, lectura, cierre del fichero.

Lectura de datos con *DataInputStream*

DataInputStream provee métodos para la lectura secuencial de tipos de datos elementales desde un fichero binario:

- *readBoolean()*: lee un valor de tipo *boolean* del fichero.
- *readByte()*: lee un valor de tipo *byte* del fichero.
- *readChar()*: lee un valor de tipo *char* del fichero.
- *readDouble()*: lee un valor de tipo *double* del fichero.
- *readFloat()*: lee un valor de tipo *float* del fichero.
- *readInt()*: lee un valor de tipo *int* del fichero.
- *readLong()*: lee un valor de tipo *long* del fichero.
- *readShort()*: lee un valor de tipo *short* del fichero.
- *readUTF()*: lee una cadena en formato UTF del fichero.

Todos estos métodos desencadenan una excepción de tipo *EOFException* si es final de fichero y no hay más datos que leer.

Observa la parte del programa donde estamos realizando la lectura secuencial de cada uno de los registros:

```
boolean eof = false;
while (!eof) {
    try {
        String pro = lector.readUTF();
        float pre = lector.readFloat();
        float uni = lector.readFloat();
        Producto p = new Producto(pro, pre, uni);
        System.out.println(p);
    } catch (EOFException e1) {
        eof = true;
    } catch (IOException e2) {
        System.out.println("Ha ocurrido un error al leer los registros");
        System.out.println(e2.getMessage());
        break; // sale del bucle while
    }
}
```

Sabemos la estructura de cada artículo y en ese orden estamos leyendo (*UTF String, float, float*). Por otro lado, declaramos la variable *eof* con el valor *false* y posteriormente le asignaremos el valor *true* cuando se produzca una excepción de tipo *EOFException*. Esta nos está permitiendo la lectura secuencial con el bucle *while* mientras *eof* no tenga el valor *true*.

Aparte de desencadenarse la excepción *EOFException* al llegar a fin de fichero, podría ocurrir algún otro tipo de error inesperado. Por esa razón hemos añadido un segundo bloque *catch*, que capturará otro tipo de *IOException*.

Lectura de teclado con InputStreamReader y BufferedReader

Hasta ahora hemos utilizado flujos de datos para leer o escribir ficheros, pero no hay que olvidar que los flujos de datos también nos permiten comunicarnos con otros dispositivos, por ejemplo, el teclado.

En este apartado utilizaremos la clase *BufferedReader* para realizar la lectura de datos desde teclado, pero antes vamos a realizar una pequeña introducción a la ya conocida clase **System**.

La clase System

Ya te has familiarizado con la clase *System* a base de incluir la sentencia *System.out.println(...)* para escribir datos en pantalla, pero nunca nos hemos parado a analizarla un poco más detenidamente. Pues bien, la clase *System*, perteneciente al paquete *java.lang*, ofrece una serie de flujos de comunicación estándares para entrada / salida. *System.in*, *System.out* y *System.err*.

- *System.in*: flujo de datos que nos permite la entrada de datos desde el teclado.
- *System.out*: flujo de datos que nos permite la salida de datos a pantalla.
- *System.err*: tiene la misma función que *System.out*, usándose generalmente para la presentación de mensajes de error.

System.out dispone de los métodos *println(...)* y *print(...)* que ya has utilizado en muchas ocasiones y que nos permiten la escritura de información en pantalla.

System.in nos abre una vía de comunicación en el teclado. En el próximo ejemplo vamos a ver cómo utilizarlo para permitir al usuario introducir su nombre por teclado utilizando un objeto *BufferedReader* como *buffer* para recibir dicho dato.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Principal {

    public static void main(String[] args) throws IOException {
        InputStreamReader teclado = new InputStreamReader(System.in);
        BufferedReader lectorTeclado = new BufferedReader(teclado);
        String nombre;
        System.out.print("Escribe aquí tu nombre: ");
        nombre = lectorTeclado.readLine();
        System.out.println("Hola "+nombre+", ¿cómo te va?");
        lectorTeclado.close();
        teclado.close();
    }
}
```

Como has podido comprobar, este programa no difiere mucho de los anteriores con sus tres pasos de apertura, lectura y cierre. La diferencia es que, en lugar de abrir un fichero, hemos abierto una vía de comunicación con un dispositivo; el teclado.

Gestión de excepciones en la lectura / escritura

Todas las excepciones que se producen en las operaciones de lectura / escritura son derivadas de *IOException*, clase base situada en *java.io*, al igual que el resto de las clases que hemos ido estudiando en esta unidad.

Recuerda el apartado anterior, cuando recorrimos secuencialmente los artículos del almacén. Retomemos ahora ese ejemplo para ver cómo se gestionaron las excepciones.

```
while (!eof) {  
    try {  
  
        String pro = lector.readUTF();  
        float pre = lector.readFloat();  
        float uni = lector.readFloat();  
        Producto p = new Producto(pro, pre, uni);  
        System.out.println(p);  
    } catch (EOFException e1) {  
        eof = true;  
    } catch (IOException e2) {  
        System.out.println("Ha ocurrido un error al leer los registros");  
        System.out.println(e2.getMessage());  
        break; // sale del bucle while  
    }  
}
```

Sabemos que se producirá una excepción de tipo *EOFException* al llegar al final de fichero, es decir, cuando se intente realizar una lectura de datos y ya no haya nada más que leer. Por ese motivo encerramos toda la operación de lectura / escritura en un *try* y capturamos en el primer *catch* la excepción *EOFException*. En otro *catch* capturamos la genérica *IOException* para que recoja cualquier otra excepción que pueda producirse. Recuerda que el orden es muy importante, ya que una *EOFException* también es una *IOException* y si intercambiamos el orden nunca llegará a ejecutarse el bloque del segundo *catch*.

Las excepciones que más utilizarás en las operaciones de lectura / escritura de fichero son: *FileNotFoundException* (fichero no encontrado) y *EOFException* (final de fichero).

Clase **IOException**

Pulsa el botón de la derecha para acceder a la página oficial de Oracle sobre la clase *IOException*.

VER

Introducción a la clase Scanner

La clase *Scanner* apareció con la versión 5 de Java para simplificar enormemente tareas de entrada o lectura de datos.

Como su nombre indica, actúa como un lector exclusivamente, no puede realizar operaciones de escritura.

Podemos leer con un objeto de la clase *Scanner* desde diversas fuentes y el origen de los datos se especifica a través del parámetro del constructor de la clase de *Scanner*.

Con un objeto de la clase *Scanner* podemos leer:

1

Desde un *String*.

`String texto = "Este es el texto que será leído por un objeto Scanner";`

`Scanner lector = new Scanner(texto);`

El origen de la lectura es un *String*.

2

Desde el teclado.

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

El origen de la lectura es el teclado.

3

Desde un fichero de texto.

`Scanner lector = new Scanner(new File("peliculas.txt"));`

El origen de la lectura es un archivo.

Scanner fragmenta los datos a leer y los va recuperando por medio de los métodos *next()*, *nextLine()*, *nextInt()*, *nextFloat()*, etc. En los siguientes apartados irás descubriendo aplicaciones prácticas del uso de la clase *Scanner*.

Lectura de cadenas de texto

La clase *Scanner* permite fragmentar un *String* y leer secuencialmente cada uno de los fragmentos.

Para dividir el *String* en fragmentos se guía por una subcadena que actúa como separador. Si no se especifica lo contrario el separador será el espacio en blanco, de modo que la lectura por defecto será palabra a palabra.

Vamos a poner en práctica varios ejemplos de lectura secuencial de un *String* con un objeto *Scanner*.

Leer un *String* palabra a palabra

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto = "La cripta mágica";
        Scanner lector = new Scanner(texto);
        while (lector.hasNext()){
            System.out.println(lector.next());
        }
        lector.close();
    }
}
```

Scanner proporciona un flujo de lectura, cuyo origen se especifica en el constructor. En este caso le hemos proporcionado una cadena con el texto "La cripta mágica", que será fragmentada por palabras. El método *lector.next()* devuelve el siguiente fragmento de la cadena de manera secuencial, por esa razón lo encerramos dentro de un *while* con la condición *lector.hasNext()*, ya que *hasNext()* devuelve *true* mientras existan más fragmentos.

Como puedes observar, volvemos a utilizar la habitual secuencia de acciones: apertura del flujo, lectura, cierre.

Leer un *String* línea a línea

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Frodo Bolsón\nSamsagaz Gamyi\nPeregrin Tuk\nM
        Scanner lector = new Scanner(texto);
        while (lector.hasNext()){
            System.out.println(lector.nextLine());
        }
        lector.close();
    }
}
```

El método *nextLine()* lee una línea completa sin fragmentarla. En la cadena *texto* hemos colocado tres veces el carácter de escape "\n" para insertar retornos de carro, generando así cuatro líneas.

Leer un *String* según separador personalizado

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Rosa López;Miguel de la Parra;Carmen Ruiz;Fra
        Scanner lector = new Scanner(texto);
        lector.useDelimiter(";");
        while (lector.hasNext()){
            System.out.println(lector.next());
        }
        lector.close();
    }
}
```

Con la sentencia `lector.useDelimiter(";");` hemos establecido el delimitador utilizado para fragmentar la cadena, de modo que habrá cuatro fragmentos para leer.

Lectura de un *String* con datos numéricos

Si la cadena que deseamos fragmentar contiene datos numéricos, es posible que nos interese recuperar los datos no como textos, sino como datos de tipo *int*, *float*, *double*, etc.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="80;25;48;56;38;46";
        Scanner lector = new Scanner(texto);
        lector.useDelimiter(";");
        while (lector.hasNext()){
            int num = lector.nextInt();
            System.out.println(num);
        }
        lector.close();
    }
}
```

En este ejemplo, la cadena contiene cantidades numéricas enteras separadas por punto y coma.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Tomates;3;1,5;Patatas;5;3,5;Pimientos;1;0,95"
        Scanner lector = new Scanner(texto);
        lector.useDelimiter(";");
        while (lector.hasNext()){
            String producto = lector.next();
            int cantidad = lector.nextInt();
            float precio = lector.nextFloat();
            float total = cantidad*precio;
            System.out.println(cantidad + " kg de " + producto
        }
        lector.close();
    }
}
```

En este ejemplo, la cadena contiene datos de ventas y hay que ir leyendo bloques de tres elementos formatos por producto (*String*), cantidad (*int*) y precio (*float*).

i **IMPORTANTE:** *nextInt()*, *nextFloat()*, *nextDouble()*, etc. podrían provocar una excepción de tipo *InputMismatchException* si el elemento leído no tiene el formato correcto para poder ser convertido al tipo numérico que corresponda.

Lectura desde teclado

En este apartado profundizaremos sobre la entrada de datos por teclado con la clase *Scanner*.

Scanner dispone de varios métodos para recibir datos desde el teclado. Comenzaremos por recordar el sistema más sencillo que ya hemos empleado en otras ocasiones, el método *nextLine()*, que recibe una línea de texto finalizada con un retorno de carro.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String nombre;
        Scanner lector = new Scanner (System.in);
        System.out.println("¿Cómo te llamas?");
        nombre = lector.nextLine();
        System.out.println("Encantado " + nombre);
        lector.close();
    }
}
```

El método *nextLine()* lee una entrada por teclado finalizada con un retorno de carro, es decir, una línea de texto. Guardamos la línea leída en la variable *nombre*.

También podemos recibir por teclado datos numéricos utilizando los métodos *nextInt()*, *nextFloat()*, *nextDouble()*, etc.

Veamos un ejemplo en el que deseamos preguntar al usuario primero su edad, para lo que utilizaremos el método *nextInt()* y después su nombre, para lo que utilizaremos el método *nextLine()*.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("¿Cuántos años tienes?");
        int edad = lector.nextInt();
        System.out.println("¿Cómo te llamas?");
        String nombre = lector.nextLine();
        System.out.println("Encantado " + nombre + " de " + edad +
lector.close();
    }
}
```

Programa que lee desde el teclado un *int* y una línea de texto.

El programa parece sencillo, sin embargo, no funciona como esperamos.

 ¿Cuántos años tienes?
18
¿Cómo te llamas?
Encantado de 18 años

Pide la edad y directamente, sin darnos opción a introducir el nombre, pasa a ejecutar la siguiente línea mostrando en pantalla "Encantado de 18 años".



¿Y por qué ocurre esto?

Durante la ejecución, al escribir la edad, no solo introdujiste un número, sino que también **pulsaste la tecla "enter"** y ahí es donde está el quid de la cuestión. La sentencia `edad = lector.nextInt()` leyó el número, pero el retorno de carro producido por la tecla "enter" quedó en el *buffer* del teclado y fue recogido por la sentencia `String nombre = lector.nextLine()`, quedando la entrada del nombre para una tercera lectura, que nunca se produjo.

Una posible solución es sustituir `nextLine()` por `next()`, pero en ese caso tendríamos problemas para recibir nombres compuestos, ya que cada lectura corresponde a una palabra.

Otra solución está en colocar un `nextLine()` detrás de cada `nextInt()`, `nextFloat()`, `nextDouble()`, etc. para que recoja el retorno de carro.

Esta última versión resuelve el problema

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("¿Cuántos años tienes?");
        int edad = lector.nextInt(); // Recibe un int
        lector.nextLine(); // Recibe el retorno de carro.
        System.out.println("¿Cómo te llamas?");
        String nombre = lector.nextLine(); // Recibe un string
        System.out.println("Encantado " + nombre + " de " + edad +
        lector.close());
    }
}
```

Lectura de ficheros de texto

En este apartado utilizaremos la clase *Scanner* para leer un fichero de texto línea a línea.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) throws FileNotFoundException
        File fichero = new File("C:\\cine\\peliculas.txt");
        if (!fichero.exists()) { // Si no existe el fichero

            System.out.println("El fichero no existe");
            return;
        }

        Scanner lector = new Scanner(fichero);
        while (lector.hasNext()) {
            String linea = lector.nextLine();
            System.out.println(linea);
        }

        lector.close();
    }
}
```

Ejecutamos sucesivas veces el método *nextLine()* mientras se cumpla la condición *lector.hasNext()*, es decir, mientras sigan quedando líneas por leer.

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Utilizamos la clase *File* para obtener **información sobre archivos y directorios**, así como crearlos, renombrarlos o eliminarlos.
- Todas las clases que representan **flujos de caracteres** en formato Unicode de 16 bits derivan directa o indirectamente de las clases abstractas *Reader* y *Writer*.
- Todas las clases que representan **flujos de bytes** (información binaria) derivan directa o indirectamente de las clases abstractas *InputStream* y *OutputStream*.
- La clase *Scanner*, disponible a partir de la versión 5 de Java, actúa como **flujo de lectura de caracteres** desde un objeto *String*, el teclado o un archivo de texto.
- *IOException* es la clase base de la que derivan todas las demás clases de excepción que tienen que ver con operaciones de entrada / salida de datos.



PROEDUCA