UD6

LENGUAJE DE PROGRAMACIÓN JAVA

MP_0485
Programación

6.1 Manejo de Ficheros

Introducción

En el desarrollo de aplicaciones profesionales, la gestión de la información no se limita únicamente al uso de bases de datos. Con frecuencia, los programas necesitan **leer, crear, modificar o almacenar datos en ficheros**, ya sea para guardar configuraciones, procesar documentos, registrar actividades o intercambiar información con otros sistemas. Por este motivo, el manejo adecuado de ficheros constituye una competencia esencial para cualquier desarrollador.

Java proporciona un conjunto amplio y robusto de herramientas para trabajar con el sistema de archivos, ofreciendo mecanismos seguros, portables y orientados a objetos. A lo largo de esta unidad didáctica, el alumnado aprenderá a utilizar las clases y paquetes que Java pone a disposición para interactuar con ficheros y directorios, desde las operaciones más básicas hasta funcionalidades más avanzadas.

En esta unidad profundizaremos en:

- **Lectura y escritura de ficheros** utilizando flujos de datos (streams) y métodos de alto nivel.
- Manejo de distintos tipos de ficheros:
  - Texto
  - Binarios
  - Serializados
  - Propiedades (.properties)
  - JSON.
- **Gestión de directorios**, rutas relativas y absolutas.
- Uso del paquete **java.io** para operaciones fundamentales.
- Control de excepciones y tratamiento de errores derivados del acceso al sistema de archivos.
- Buenas prácticas de programación, como:
  - Cierre correcto de flujos
  - Manejo de recursos con *try-with-resources*
  - Validación de rutas
  - Prevención de pérdidas de datos

Comprender cómo se gestionan los ficheros en Java permitirá a los estudiantes desarrollar aplicaciones más completas y profesionales, capaces de almacenar y persistir información

independientemente del entorno en el que se ejecuten. Además, se sentarán las bases necesarias para unidades posteriores en las que se abordará la persistencia mediante bases de datos y otros sistemas de almacenamiento.

Esta unidad será, por tanto, un paso fundamental para que el alumnado adquiera autonomía en la creación de aplicaciones reales, ya sea en proyectos de clase, en el módulo de empresa o en futuros entornos laborales.

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:

- File (String path).
- File (String path, String nameFile or path).
- File (File path, String nameFile or path).

Donde Path representa la ruta dentro de la estructura de carpetas y podrá especi carse 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.

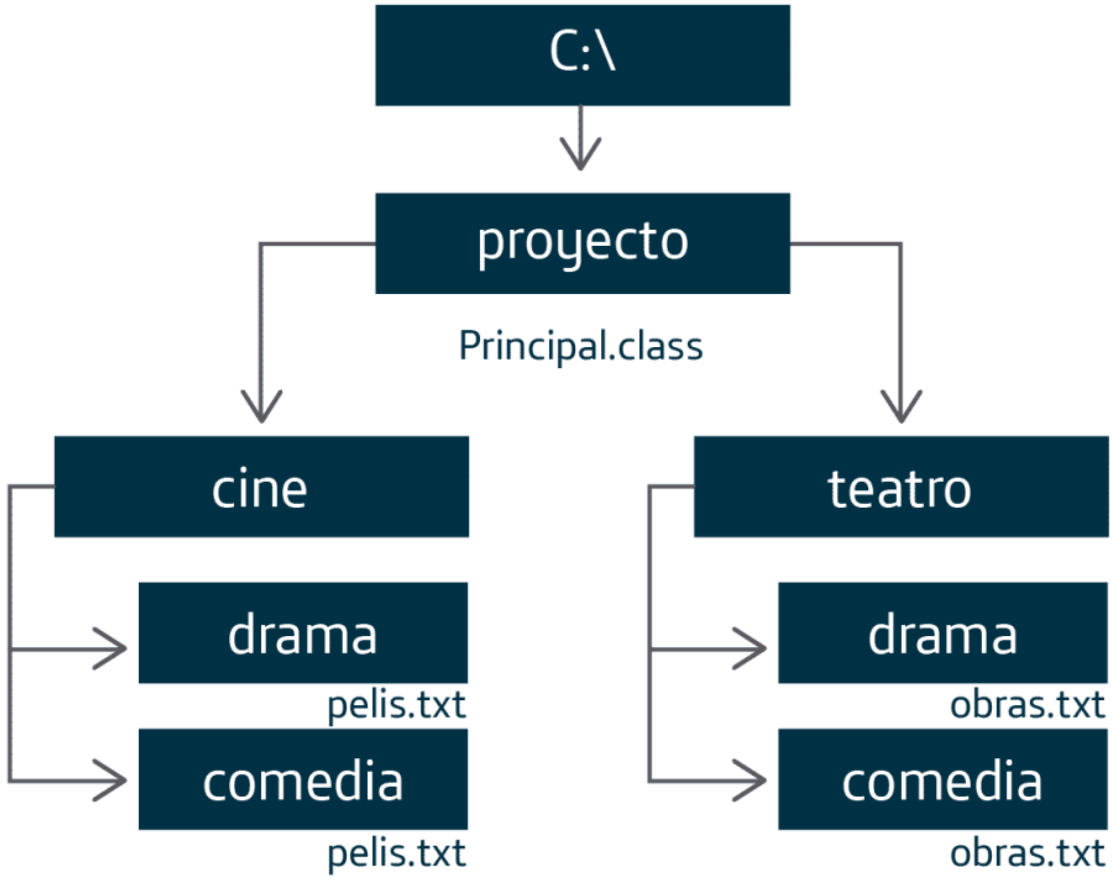


Figura 1: Diagrama de una estructura de carpetas y ficheros

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

Vamos a ver unos ejemplos del uso de la clase `File`.

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

El objeto `fich` representa el chero 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).`

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
- 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

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 ch representa el chero 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 chero. 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:

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

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

Sin embargo, **la clase `File` NO permite**:

- 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.

- 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

Vamos 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 Main {
    public static void main(String[] args) {
        File fich = new File( pathname: "C:/proyecto/cine/drama/pelis.txt");
        if (fich.exists()) {
            System.out.println("Existe el fichero");
            System.out.println("N° 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. Vamos a ver como podemos hacerlo.

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

public class Main {
    public static void main(String[] args)  {
        File fich = new File( pathname: "C:/proyecto/cine/drama/pelisdeterror.txt");
        boolean ok = false;
        try {
            ok = fich.createNewFile();
        } catch (IOException e) {
            System.out.println("La ruta especificada no existe");
        }
        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.

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

```
import java.io.File;

public class Main {
    public static void main(String[] args) {
        File fich = new File( pathname: "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");
    }
}
```

Obteniendo información de una carpeta

La clase File también nos permite trabajar con carpetas.

```
import java.io.File;

public class Main {
    public static void main(String[] args) {
        File carp = new File( pathname: "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.

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.

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

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 acceder a una carpeta que tenga más contenido, por ejemplo la carpeta Windows, que casi seguro está situada en C:\Windows.

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.

```java
public class Main {
    public static void main(String[] args) {
        File carp = new File( pathname: "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: "+f.getTotalSpace());
            }
        }
        else
            System.out.println("No existe la carpeta");
    } }
```

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

```java
import java.io.File;

public class Main {
    public static void main(String[] args) {
        File carp = new File( pathname: "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");
    }
}
```

Ahora vamos a borrar la carpeta que acabamos de crear. 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.

```
import java.io.File;

public class Main {
    public static void main(String[] args) {
        File carp = new File( pathname: "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");
    }
}
```

Clases que representan flujos de datos

Toda la información que se trasmite 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.

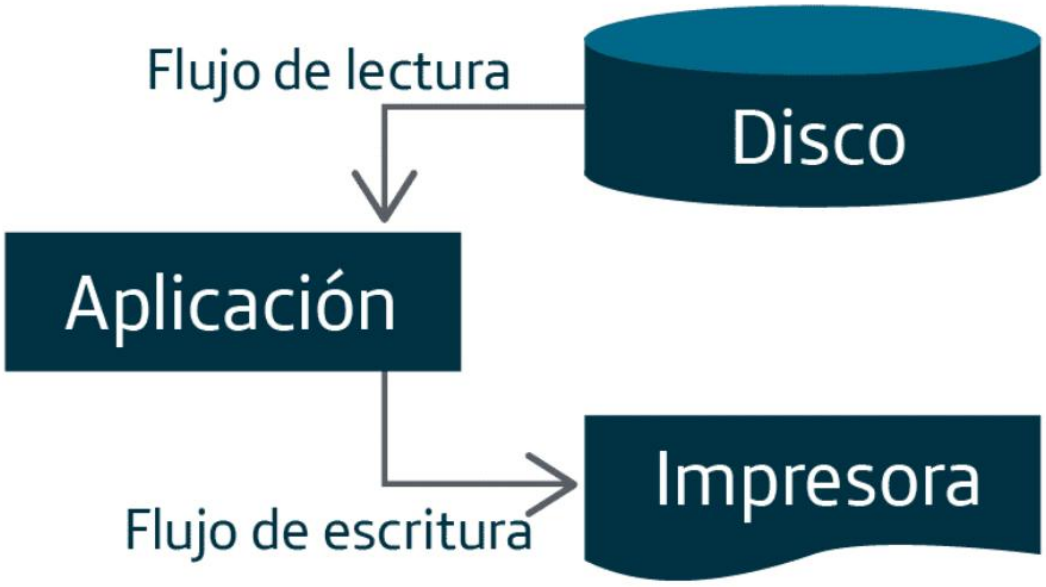
Los stream permiten transmitir secuencias ordenadas de datos desde un origen a un destino. El origen y el destino puede ser un fichero, un String o un dispositivo (lectura de teclado, escritura en pantalla).

Java dispone de dos grupos de flujos de datos:

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

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

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



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

- Abrir el flujo de datos de lectura o de escritura.
- Leer o escribir datos a través del flujo abierto.
- 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:

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

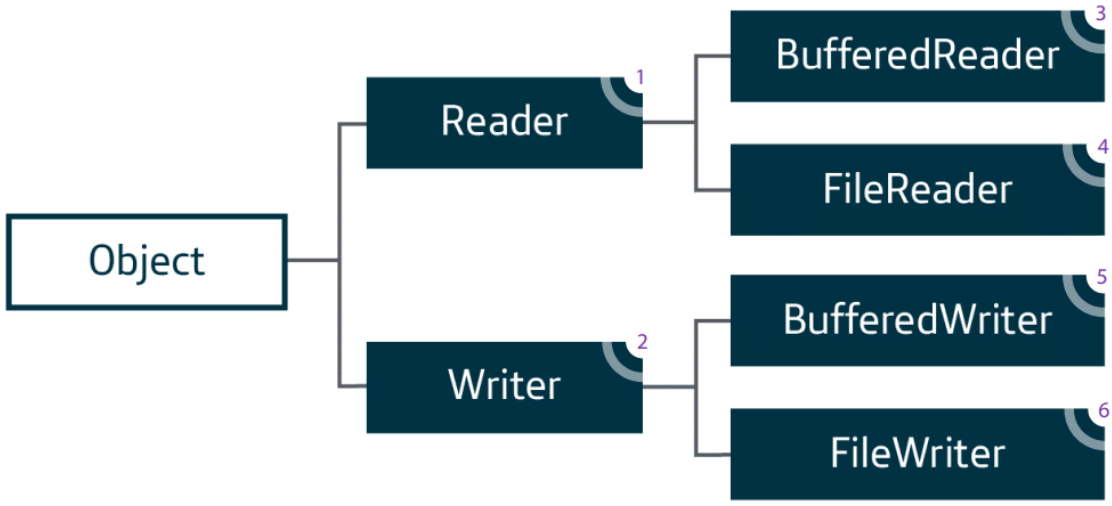
- **Flujos de bytes (información binaria)**: derivados de las clases abstractas `InputStream` y `OutputStream`.

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.

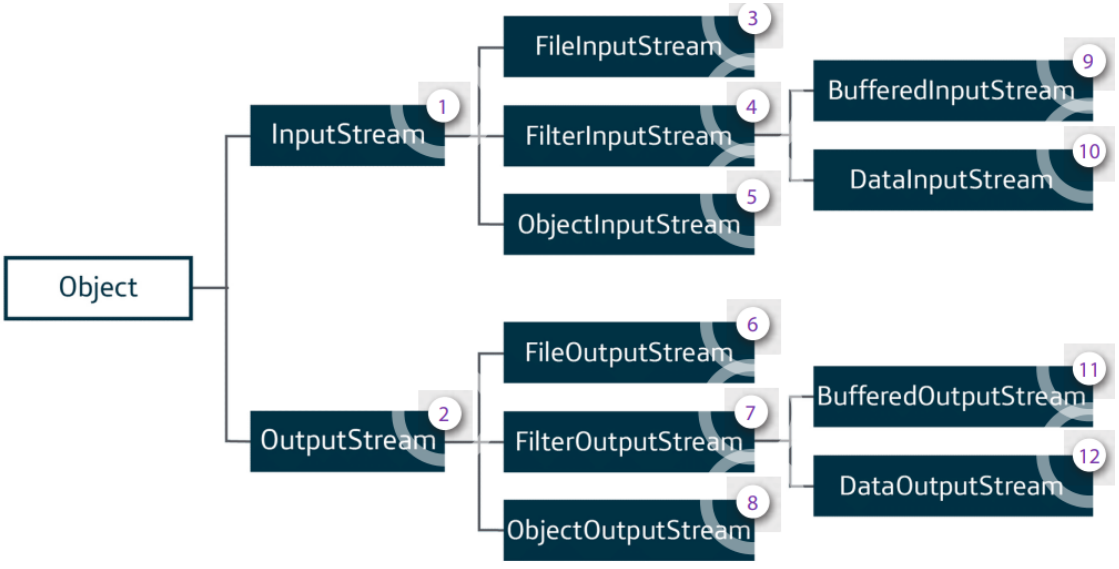
5. **BufferedWriter**: Permite mejorar la velocidad de escritura en un chero proporcionando un buffer. Entra dentro de la categoría de filtro y trabaja en colaboración con la clase FileWriter.

6. **FileWriter**: Permite escribir caracteres en un chero. 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 chero. 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 chero. 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 e ciencia de la lectura de un chero proporcionando un buffer. Trabaja en colaboración con un objeto iniciador, por ejemplo, un `FileInputStream`.

10. **DataInputStream:** `DataInputStream` permite leer datos de un chero 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. **BufferedOutStream:** Permite mejorar la eficiencia de la escritura en un chero proporcionando un buffer. Trabaja en colaboración con un objeto iniciador, por ejemplo, un `FileOutputStream`.

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

Referencias

Apuntes elaborados a partir de la siguiente documentación:

[1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.

[2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.

[3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.

[4] Apuntes de Programación Edix Digital Workers.