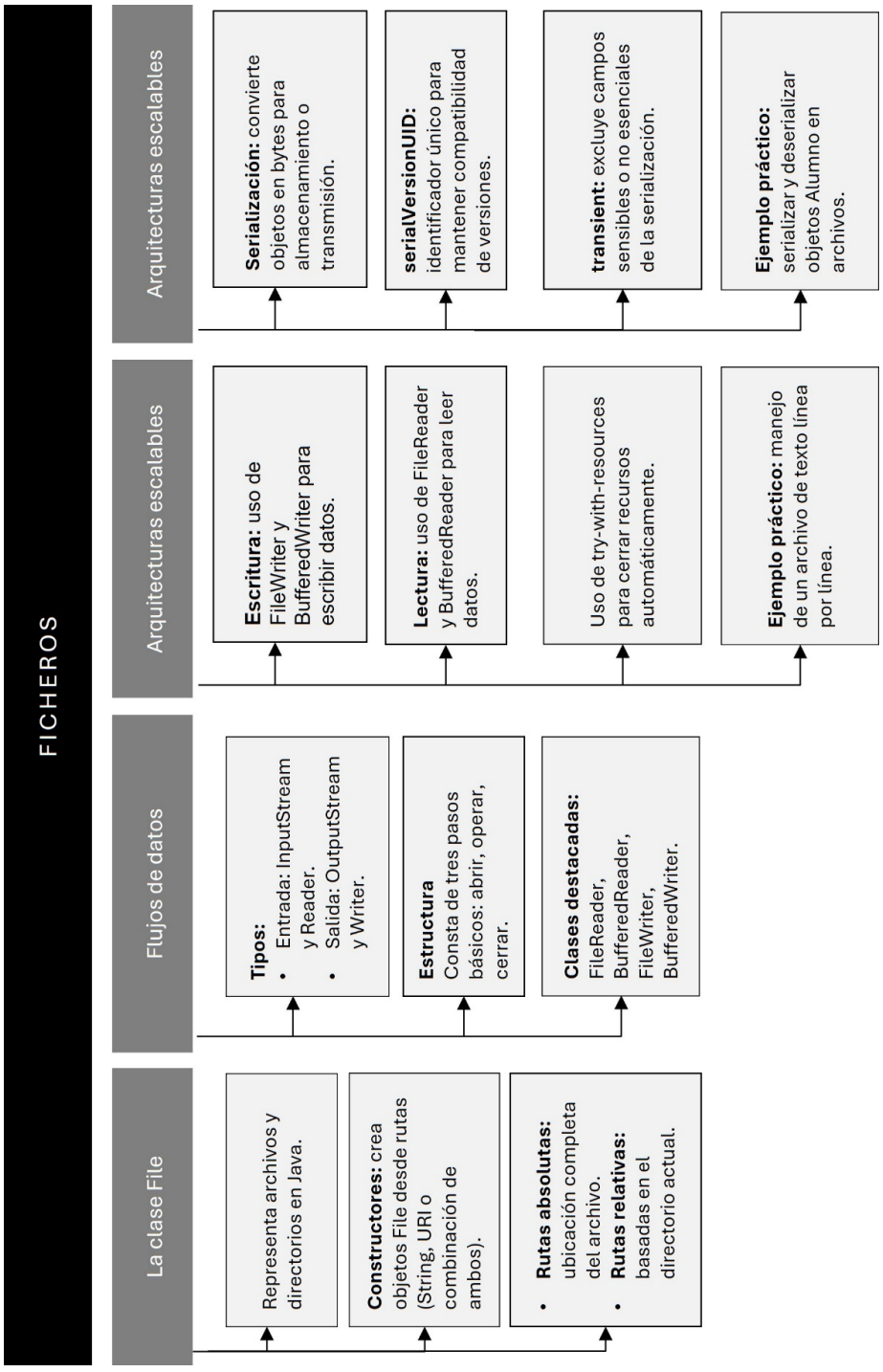
Acceso a Datos

Tema 1. Ficheros

Índice


FICHEROS
La clase File
Flujos de datos
Arquitecturas escalables
Arquitecturas escalables
Representa archivos y directorios en Java.
Constructores: crea objetos File desde rutas (String, URI o combinación de ambos).
Rutas absolutas: ubicación completa del archivo.
Rutas relativas: basadas en el directorio actual.
Tipos:
Entrada: InputStream y Reader.
Salida: OutputStream y Writer.
Estructura
Consta de tres pasos básicos: abrir, operar, cerrar.
Clases destacadas: FileReader, BufferedReader, FileWriter, BufferedWriter.
Escritura: uso de FileWriter y BufferedWriter para escribir datos.
Lectura: uso de FileReader y BufferedReader para leer datos.
Uso de try-with-resources para cerrar recursos automáticamente.
Ejemplo práctico: manejo de un archivo de texto línea por línea.
Serialización: convierte objetos en bytes para almacenamiento o transmisión.
serialVersionUID: identificador único para mantener compatibilidad de versiones.
transient: excluye campos sensibles o no esenciales de la serialización.
Ejemplo práctico: serializar y deserializar objetos Alumno en archivos.

1.1. Introducción y objetivos

En el **desarrollo de aplicaciones multiplataforma,** la gestión eficiente de archivos es esencial para el almacenamiento y recuperación de datos. Este tema aborda las técnicas y herramientas fundamentales para manipular archivos en entornos de programación, proporcionando a los estudiantes las habilidades necesarias para manejar información de manera efectiva.

Comienza con una introducción a la clase `File`, que permite interactuar con el sistema de archivos del entorno de ejecución. Se exploran métodos para crear, eliminar y obtener información sobre archivos y directorios, estableciendo una base sólida para operaciones más complejas. Posteriormente, se abordan las técnicas de lectura y escritura en archivos de texto, esenciales para el procesamiento de datos almacenados en este formato.

Además, se introduce la interfaz serializable, que facilita la conversión de objetos en una secuencia de bytes para su almacenamiento o transmisión. Los estudiantes aprenderán a grabar y recuperar objetos desde el disco, comprendiendo la importancia del modificador `transient` para controlar la serialización de ciertos atributos. Estos conocimientos son fundamentales para el desarrollo de aplicaciones que requieren persistencia de datos de manera eficiente y segura.

los objetivos que se pretenden conseguir en este tema son:

- Comprender el uso de la clase `File` para la gestión de archivos y directorios en aplicaciones multiplataforma.
- Aprender a realizar operaciones de lectura y escritura en archivos de texto, manejando adecuadamente los flujos de entrada y salida.
- Entender la interfaz `Serializable` y su aplicación en la serialización y deserialización de objetos.

- Implementar técnicas para almacenar y recuperar objetos desde el disco, garantizando la integridad y consistencia de los datos.
- Conocer el uso del modificador `transient` y su impacto en el proceso de serialización de objetos.

1.2. La clase File

La clase `File` de Java, ubicada en el paquete java.io, representa abstracciones de archivos y directorios en el sistema de archivos. Aunque no permite la lectura o escritura directa de datos, proporciona métodos para manipular y obtener información sobre archivos y directorios, como crear, eliminar, verificar existencia y listar contenidos.

Constructores de la clase File

La clase File ofrece varios constructores para crear instancias que representan rutas específicas:

- **File(String pathname)**: crea una instancia que representa la ruta especificada por `pathname` . Este puede ser una ruta absoluta o relativa.

```
File archivo = new File("ruta/al/archivo.txt");
```

- **File(String parent, String child)**: combina una ruta de directorio padre con un nombre de archivo o subdirectorio hijo.

```
File archivo = new File("ruta/al/directorio", "archivo.txt");
```

- **File(File parent, String child)**: similar al anterior, pero el directorio padre se especifica como un objeto `File` .

```
File directorioPadre = new File("ruta/al/directorio");

File archivo = new File(directorioPadre, "archivo.txt");
```

- **File(URI uri)**: crea una instancia a partir de un objeto `URI`. Este constructor es útil cuando se trabaja con rutas en formato `URI`.

```
URI uri = new URI("file:///ruta/al/archivo.txt");

File archivo = new File(uri);
```

Rutas absolutas y relativas

Al utilizar la clase `File`, es importante comprender el contexto del directorio de trabajo actual, especialmente cuando se emplean rutas relativas, para asegurar que las operaciones apunten a las ubicaciones correctas en el sistema de archivos.

- **Ruta absoluta**: especifica la ubicación completa del archivo o directorio desde la raíz del sistema de archivos. Por ejemplo:

  - En sistemas Unix: `/home/usuario/archivo.txt`
  - En Windows: `C:\\Usuarios\\usuario\\archivo.txt`.

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

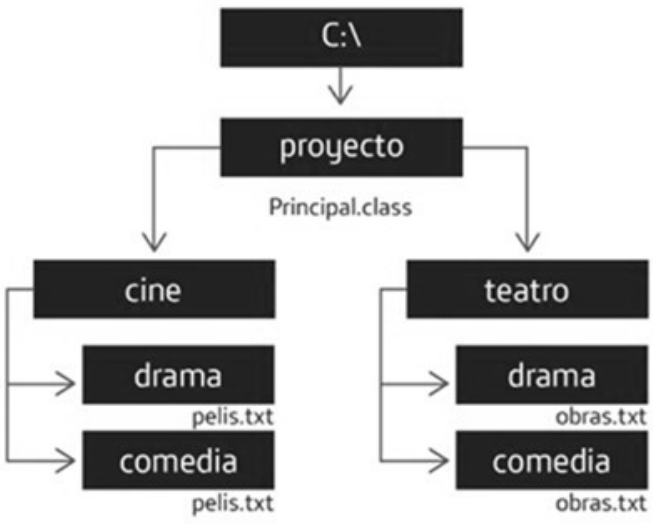


Figura 1. Estructura de directorios y archivos. Fuente: elaboración propia.

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 cheros de texto.

Al utilizar la clase `File`, es importante comprender el **contexto del directorio de trabajo actual,** especialmente cuando se emplean rutas relativas, para asegurar que las operaciones apunten a las ubicaciones correctas en el sistema de archivos.

Para **obtener la ruta absoluta de un archivo** representado por un objeto `File`, se puede utilizar el método `getAbsolutePath()`.

```
File archivo = new File("archivo.txt");

String rutaAbsoluta = archivo.getAbsolutePath();

System.out.println("Ruta absoluta: " + rutaAbsoluta);
```

Este método devuelve la ruta completa del archivo, resolviendo cualquier referencia relativa en función del directorio de trabajo actual.

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

El objeto `fichero` representa el fichero especificado en el argumento. Se ha utilizado el primer constructor.

```
File fichero = 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 *ch*, que representa un chero 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).*

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. De esta forma, no tenemos errores de compilación.

- **Ruta relativa**: especifica la ubicación en relación con el directorio de trabajo actual de la aplicación. Por ejemplo, ./archivo.txt se refiere a un archivo en el directorio actual, mientras que ../archivo.txt apunta al archivo en el directorio padre.

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 ch, que representa un chero a partir del objeto carp, que representa la carpeta donde está ubicado el chero. Estamos utilizando el **tercer constructor** para crear el objeto ch: File (File path, String nameFile).

Acciones que se pueden llevar a cabo con el objeto File

Son muchas las acciones que se pueden llevar a cabo con este tipo de objetos; las más conocidas son:

1. Crear nuevos archivos y directorios

- Crear un archivo nuevo:

```
import java.io.File;

import java.io.IOException;

public class CrearArchivo {

    public static void main(String[] args) {

        File archivo = new File("nuevoArchivo.txt");

        try {

            if (archivo.createNewFile()) {

                System.out.println("Archivo creado: " + archivo.getName());

            } else {

                System.out.println("El archivo ya existe.");

            }

        } catch (IOException e) {

            System.out.println("Ocurrió un error.");

            e.printStackTrace();

        }

    }
```

```
}
```

- Crear un directorio nuevo:

```
import java.io.File;

public class CrearDirectorio {

    public static void main(String[] args) {

        File directorio = new File("nuevoDirectorio");

        if (directorio.mkdir()) {

            System.out.println("Directorio creado: " +
directorio.getName());

        } else {

            System.out.println("No se pudo crear el directorio.");

        }

    }

}
```

2. Comprobar la existencia, legibilidad y capacidad de escritura de archivos

Verificar si un archivo existe y si es legible y escribible:

```
import java.io.File;

public class VerificarArchivo {

    public static void main(String[] args) {

        File rchive = new File(“rchive.txt”);
```

```
        if (rchive.exists()) {
            System.out.println(“El rchive existe.”);
            System.out.println(“¿Es legible? “ + rchive.canRead());
            System.out.println(“¿Es escribible? “ + rchive.canWrite());
        } else {
            System.out.println(“El rchive no existe.”);
        }
    }
}
```

3. Obtener información sobre archivos y directorios

Obtener el nombre, la ruta absoluta y el tamaño de un archivo:

```
import java.io.File;
public class InformacionArchivo {
    public static void main(String[] args) {
        File archivo = new File("archivo.txt");
        if (archivo.exists()) {
            System.out.println("Nombre del archivo: " + archivo.getName());
            System.out.println("Ruta absoluta: " +
archivo.getAbsolutePath());
            System.out.println("Tamaño: " + archivo.length() + " bytes");
        } else {
```

```
            System.out.println("El archivo no existe.");
        }
    }
}
```

4. Listar el contenido de un directorio

Listar los nombres de archivos y subdirectorios en un directorio:

```
import java.io.File;
public class ListarDirectorio {
    public static void main(String[] args) {
        File directorio = new File("ruta/al/directorio");
        if (directorio.isDirectory()) {
            String[] contenido = directorio.list();
            if (contenido != null) {
                for (String nombre : contenido) {
                    System.out.println(nombre);
                }
            } else {
                System.out.println("No se pudo obtener el contenido del
directorio.");
            }
```

```
        } else {
            System.out.println("La ruta especificada no es un
directorio.");
        }
    }
}
```

5. Eliminar archivos y directorios

- Eliminar un archivo:

```
import java.io.File;
public class EliminarArchivo {
    public static void main(String[] args) {
        File archivo = new File("archivo.txt");
        if (archivo.delete()) {
            System.out.println("Archivo eliminado: " + archivo.getName());
        } else {
            System.out.println("No se pudo eliminar el archivo.");
        }
    }
}
```

- Eliminar un directorio vacío:

```
import java.io.File;

public class EliminarDirectorio {

    public static void main(String[] args) {

        File directorio = new File("directorioVacio");

        if (directorio.delete()) {

            System.out.println("Directorio eliminado: " +
directorio.getName());

        } else {

            System.out.println("No se pudo eliminar el directorio.
Asegúrese de que esté vacío.");

        }

    }

}
```

Estos ejemplos ilustran algunas de las operaciones básicas que se pueden realizar utilizando la clase `File` en Java. Es importante manejar adecuadamente las **excepciones** y verificar las **condiciones necesarias** al interactuar con el sistema de archivos para garantizar un funcionamiento correcto de las aplicaciones.

1.3. 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.

Figura 2. Representación flujo de datos. Fuente: elaboración propia.

Los `Stream` permiten transmitir secuencias ordenadas de datos desde un origen a un destino. El origen y el destino puede ser un chero, 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 chero o dispositivo hacia el programa.
- **Flujos de salida o escritura *(output streams)*:** los datos fluyen desde el programa hacia el chero o dispositivo.

`Java` **no dispone de flujos de entrada / salida.** No cuenta con clases que representen flujos de lectura y escritura. Si necesitamos leer y escribir de un chero, requeriremos dos flujos distintos: un flujo de entrada o lectura y otro de salida o escritura.

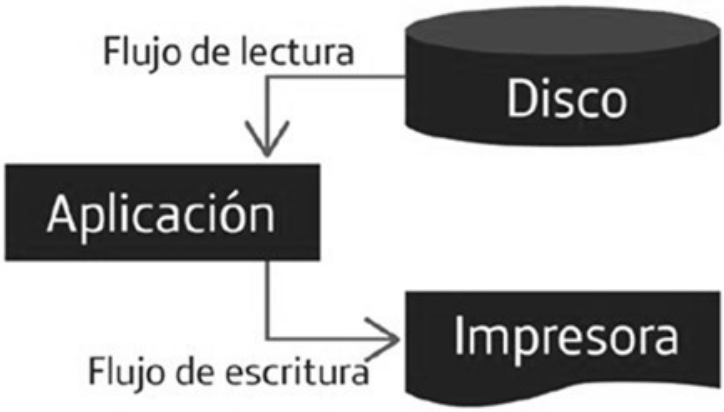


Figura 3. Organigrama de lectura y escritura de archivos. Fuente: elaboración propia.

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.

Con respecto a 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 se han 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.

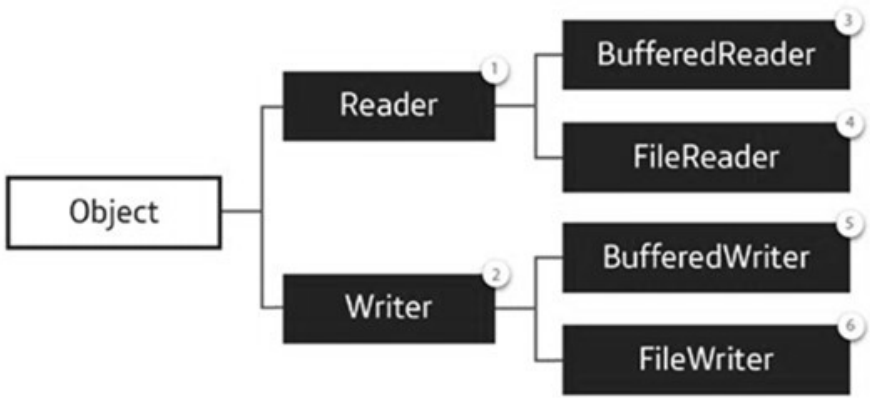


Figura 4. Organigrama clases de Streams en Java. Fuente: elaboración propia.

Reader

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

- **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 .

- **FileReader:** permite leer caracteres de un chero. Es iniciador y puede trabajar en conjunto con la clase BufferedReader , que actúa como filtro y mejora la eficiencia de la lectura.

Writer

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

- **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 .

- **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 InputStreamy OutputStream. En la imagen se ha 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.

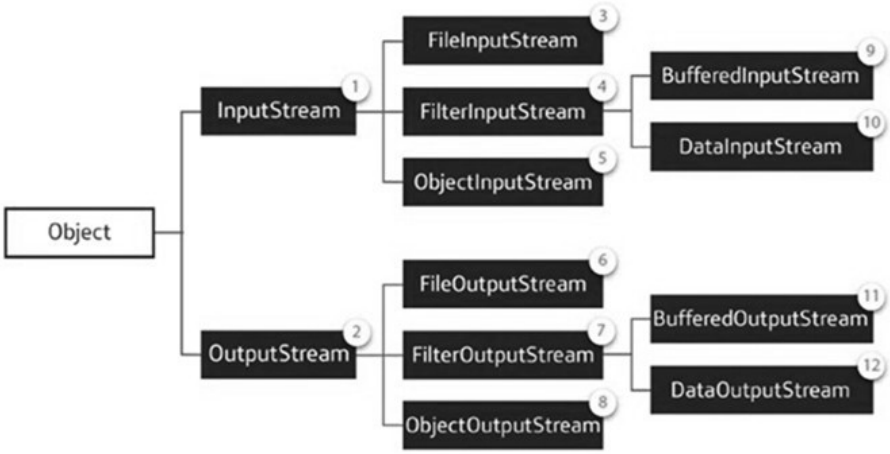


Figura 5. Organigrama clases de Streams de información binaria en Java. Fuente: elaboración propia.

InputStream

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

- **FileInputStream:** permite leer *bytes* de un chero. Actúa como iniciador.
- **FilterInputStream:** clase base de la que derivan las siguientes subclases que actúan como ltro , mejorando las operaciones de lectura:
  - **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 .
  - **DataInputStream:** permite leer datos de un chero recogiéndolos directamente como tipos de datos primitivos (*int*, *oat*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como FileInputStream.
- **ObjectInputStream:** permite la lectura de objetos guardados en disco. Actúa como filtro y requiere un iniciador, por ejemplo, un FileInputStream.

OutputStream

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

- **FileOutputStream:** permite escribir *bytes* en un chero. Actúa como iniciador.
- **FilterOutputStream:** clase base de la que derivan las siguientes subclases que actúan como filtro, mejorando las operaciones de escritura:
  - **BufferedOutStream:** permite mejorar la e-ciencia de la escritura en un chero proporcionando un buffer . Trabaja en colaboración con un objeto iniciador, por ejemplo, un FileOutputStream.
  - **DataOutputStream:** permite escribir datos en un chero directamente como tipos de datos primitivos (*int*, *oat*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como FileOutputStream.
- **ObjectOutputStream:** permite la escritura de objetos en disco. Actúa como filtro y requiere un iniciador, por ejemplo, un FileOutputStream.

1.4. Escritura en fichero de texto

Generalmente, las **operaciones de lectura y 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 chero de texto. Para ello, realizaremos los **tres pasos** típicos asociados a cualquier tarea de escritura en ficheros:

- Abrir fichero para escritura.
- Escribir líneas en el fichero.
- Cerrar el fichero.

```
import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class EscribirPeliculas {

    public static void main(String[] args) {

        // Ruta del archivo donde se escribirán los nombres de las
películas

        String nombreArchivo = "peliculas.txt";

        // Array con los nombres de las películas
```

```
        String[] peliculas = {"El Padrino", "Pulp Fiction", "La La Land"};

        // Uso de try-with-resources para asegurar el cierre de los
recursos

        try (BufferedWriter escritor = new BufferedWriter(new
FileWriter(nombreArchivo))) {

            // Escribir cada película en una nueva línea

            for (String pelicula : peliculas) {

                escritor.write(pelicula);

                escritor.newLine(); // Añade un salto de línea

            }

            System.out.println("Las películas se han escrito correctamente
en el archivo.");

        } catch (IOException e) {

            System.err.println("Ocurrió un error al escribir en el archivo:
" + e.getMessage());

        }

    }

}
```

Ahora vamos a analizar las partes más importantes del programa:

- **Importación de clases necesarias**: se importan `BufferedWriter`, `FileWriter` y `IOException` del paquete java.io.
- **Definición del nombre del archivo**: se especifica el nombre del archivo donde se almacenarán los nombres de las películas. En este caso, peliculas.txt.

- **Array de películas**: se crea un array de cadenas que contiene los nombres de las tres películas a escribir.
- **Estructura try-with-resources**: esta estructura garantiza que los recursos se cierren automáticamente al finalizar el bloque, incluso si ocurre una excepción. Se crea una instancia de `BufferedWriter` que envuelve a un `FileWriter`.
- **Escritura en el archivo**: se itera sobre el array de películas, escribiendo cada nombre en una nueva línea del archivo utilizando el método `write()` de `BufferedWriter`. Después de cada escritura, se llama a `newLine()` para insertar un salto de línea.
- **Manejo de excepciones**: se captura cualquier `IOException` que pueda ocurrir durante la escritura y se muestra un mensaje de error con el detalle de la excepción.

Este enfoque es eficiente y asegura que los recursos se gestionen adecuadamente. De esta forma, evita posibles fugas de recursos y garantiza que el archivo se cierre correctamente después de la operación de escritura.

La sentencia:

```
new FileWriter(nombreArchivo)
```

No solo nos ha permitido abrir el chero 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 chero existente, solo tenemos que pasar un argumento más al constructor de `FileWriter` con el valor true.

```
new FileWriter(nombreArchivo, true)
```

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

1.5. Lectura de fichero de texto

Para leer el contenido del archivo peliculas.txt, que contiene los nombres de tres películas, utilizaremos las clases FileReader (iniciador) y BufferedReader (filtro) del paquete java.io. Estas clases permiten leer archivos de texto de manera eficiente, porque facilitan la lectura línea por línea.

A continuación, se presenta un ejemplo de cómo leer el archivo y mostrar su contenido en la consola:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class LeerPeliculas {

public static void main(String[] args) {

  // Nombre del archivo a leer

String nombreArchivo = "peliculas.txt";

// Uso de try-with-resources para asegurar el cierre de los recursos

  try (BufferedReader lector = new BufferedReader(new FileReader(nombreArchivo))) {

  String linea;

// Leer cada línea del archivo hasta el final

  while ((linea = lector.readLine()) != null) {
```

```
System.out.println(linea);

}

} catch (IOException e) {

 System.err.println("Ocurrió un error al leer el archivo: " + e.getMessage());

 }

}

}
```

Explicación del código:

- **Importación de clases necesarias**: se importan `BufferedReader`, `FileReader` y `IOException` del paquete java.io.
- **Definición del nombre del archivo**: se especifica el nombre del archivo que se desea leer, en este caso, peliculas.txt.
- **Estructura try-with-resources**: esta estructura garantiza que los recursos se cierren automáticamente al finalizar el bloque, incluso si ocurre una excepción. Se crea una instancia de `BufferedReader` que envuelve a un `FileReader`.
- **Lectura del archivo línea por línea**: se utiliza un bucle `while` que llama al método `readLine()` de `BufferedReader` para leer cada línea del archivo. Este método devuelve `null` cuando se alcanza el final del archivo. Cada línea leída se almacena en la variable `linea` y se imprime en la consola.
- **Manejo de excepciones**: se captura cualquier IOException que pueda ocurrir durante la lectura y se muestra un mensaje de error con el detalle de la excepción.

1.6. La interfaz serializable

En Java, la **serialización** es el proceso que permite convertir un objeto en una secuencia de *bytes,* lo que facilita su almacenamiento en archivos o su transmisión a través de redes. La deserialización es el proceso inverso, donde se reconstruye el objeto original a partir de dicha secuencia de bytes. Para que una clase sea serializable, debe implementar la interfaz `Serializable` del paquete java.io. Esta interfaz actúa como una interfaz de marcador *(marker interface),* es decir, no contiene métodos ni campos, pero indica al sistema que los objetos de la clase pueden ser serializados.

Uso de la interfaz Serializable

Al implementar `Serializable` , se habilita la serialización automática de los objetos de la clase. Por ejemplo:

```
import java.io.Serializable;

public class Persona implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nombre;

    private int edad;

    // Constructor, getters y setters

}
```

En este ejemplo, la clase `Persona` es serializable, lo que permite que sus objetos se conviertan en una secuencia de *bytes* para su almacenamiento o transmisión.

Importancia del serialVersionUID

El campo, serialVersionUID es un identificador único que se utiliza durante la deserialización para verificar que el emisor y el receptor de un objeto serializado han cargado clases compatibles en términos de serialización. Si no coinciden, se lanzará una excepción InvalidClassException.

Aunque el **entorno de ejecución** de Java puede calcular un serialVersionUID predeterminado, basado en diversos aspectos de la clase, se recomienda **declarar este campo explícitamente** para asegurar la compatibilidad entre diferentes versiones de la clase y evitar posibles problemas durante la deserialización.

¿Y cuándo se produce un problema de incompatibilidad? Lo comprenderás mejor con un ejemplo:

- Tenemos un programa A con una clase Alumno , que cuenta con las propiedades nombre, edad y teléfono. Dentro de la clase Principal creamos un objeto Alumno y lo guardamos en un archivo llamado datos.dat. El programa A es el que realiza la serialización y, por lo tanto, el emisor del objeto guardado.

- Tenemos otro programa B, donde hemos copiado la clase Alumno , pero esta vez se nos ha ocurrido añadir una propiedad más llamada domicilio . Los programas A y B tienen distinta versión de la clase Alumno .

Ahora, desde la clase Principal del programa B recuperamos el objeto Alumno que previamente guardamos en el archivo datos.dat durante la ejecución del programa A. El programa B debe realizar la deserialización del objeto guardado y, por lo tanto, será el receptor de dicho objeto.

El programa B nos arroja una **excepción,** ya que intenta recuperar un objeto construido a partir de la primera versión de la clase `Alumno`, sin embargo, el programa B contiene la segunda versión de la clase `Alumno`, que resulta incompatible.

Las versiones primera y segunda de la clase `Alumno` deberían tener distinto `serialVersionUID` para poder distinguir rápidamente que se trata de versiones distintas de la misma clase. De esta forma, en el proceso de deserialización, la máquina virtual de Java comparará la `serialVersionUID` del objeto guardado con la `serialVersionUID` de la clase `Alumno` que contiene el programa B, arrojando una excepción de tipo `InvalidClassException`.

¿Y qué pasa si no declaramos la serialVersionUID?

Si dentro de la clase no hemos especificado un valor de `serialVersionUID`, la máquina virtual de Java debe examinar las clases de origen y destino generando las `serialVersionUID` de forma dinámica. Aunque no sea obligatorio, resulta mucho más rápido y efectivo declarar la `serialVersionUID` y modificarla en cada nueva versión.

1.7. Grabar y recuperar un objeto de disco

En este apartado veremos cómo grabar un objeto de la clase Alumno en un archivo.

Empezamos creando la clase Alumno :

```
import java.io.*;

// Clase Alumno implementando Serializable

class Alumno implements Serializable {

    private static final long serialVersionUID = 1L; // serialVersionUID
recomendado

    private String nombre;

    private int edad;

    private String matricula;

    // Constructor

    public Alumno(String nombre, int edad, String matricula) {

        this.nombre = nombre;

        this.edad = edad;

        this.matricula = matricula;

    }

    // Método toString para imprimir la información del alumno

    @Override

    public String toString() {
```

```
        return "Alumno{" +
                "nombre='" + nombre + '\'' +
                ", edad=" + edad +
                ", matricula='" + matricula + '\'' +
                '}';
    }
}
```

Ahora procedemos a crear los objetos, guardarlos en un fichero y finalmente leerlos.

```
public class GuardarAlumno {
    public static void main(String[] args) {
        // Crear un objeto Alumno
        Alumno alumno = new Alumno("María López", 21, "A12345");
        // Nombre del archivo donde se guardará el objeto
        String nombreArchivo = "alumno.dat";
        // Guardar el objeto Alumno en un archivo
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(nombreArchivo))) {
            oos.writeObject(alumno);
            System.out.println("Objeto Alumno guardado correctamente en el
archivo.");
        } catch (IOException e) {
```

```
            System.err.println("Ocurrió un error al guardar el objeto: " +
e.getMessage());

        }

        // Leer el objeto Alumno del archivo

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(nombreArchivo))) {

            Alumno alumnoLeido = (Alumno) ois.readObject();

            System.out.println("Objeto Alumno leído del archivo:");

            System.out.println(alumnoLeido);

        } catch (IOException | ClassNotFoundException e) {

            System.err.println("Ocurrió un error al leer el objeto: " +
e.getMessage());

        }

    }

}
```

Explicación del código:

- **Clase** Alumno **:** implementa Serializable para permitir la serialización. Incluye un serialVersionUID para mantener la compatibilidad de versiones de la clase. Define atributos, constructor y un método toString para facilitar la impresión de los datos.
- **Guardar el objeto**: se usa un ObjectOutputStream que envuelve un FileOutputStream para escribir el objeto serializado en el archivo alumno.dat. El método writeObject serializa el objeto y lo escribe en el archivo.

- **Leer el objeto**: se usa un `ObjectInputStream` que envuelve un `FileInputStream` para leer el objeto del archivo. El método `readObject` deserializa los datos del archivo y reconstruye el objeto original.
- **Uso de try-with-resources**: asegura que los flujos de entrada/salida se cierren automáticamente, incluso si ocurre una excepción.

Salida esperada: cuando ejecutes este programa, debería mostrar algo similar a lo siguiente en la consola:

```
Objeto Alumno guardado correctamente en el archivo.

Objeto Alumno leído del archivo:

Alumno{nombre='María López', edad=21, matricula='A12345'}
```

1.8. El modificador transient

El **modificador** `transient` en Java se utiliza para indicar que un campo de una clase no debe ser serializado cuando el objeto se convierta en una secuencia de *bytes.* Esto significa que el valor del campo `transient` no se guardará cuando el objeto se serialice y su valor se establecerá en el valor predeterminado de su tipo de datos (por ejemplo, `null` para objetos, para tipos numéricos, `false` para booleanos) al deserializarlo.

El modificador `transient` se utiliza para excluir ciertos campos de la serialización. Esto es útil cuando ciertos datos no deben persistir o se pueden recalcular fácilmente.

¿Para qué sirve el modificador transient?

- **Evitar serialización de datos sensibles**: se utiliza para proteger información confidencial o sensible que no debería ser persistida, como contraseñas o claves.
- **Omitir datos no relevantes**: es útil para excluir campos que no sean necesarios para la reconstrucción del objeto, como datos calculados o dependientes de otros campos.
- **Optimización**: reduce el tamaño del archivo serializado al omitir campos innecesarios.

Aplicación del modificador transient en el ejemplo anterior

Supongamos que queremos agregar un campo `notaMedia` al objeto `Alumno`, pero no queremos que este campo se serialice porque se puede calcular a partir de otros datos. Aquí está el ejemplo modificado:

Código actualizado

```
import java.io.*;

// Clase Alumno con un campo transient

class Alumno implements Serializable {

    private static final long serialVersionUID = 1L; // serialVersionUID
recomendado

    private String nombre;

    private int edad;

    private String matricula;

    private transient double notaMedia; // Campo que no se serializa

    // Constructor

    public Alumno(String nombre, int edad, String matricula, double
notaMedia) {

        this.nombre = nombre;

        this.edad = edad;

        this.matricula = matricula;

        this.notaMedia = notaMedia;

    }
```

```
    // Método toString para imprimir la información del alumno
    @Override
    public String toString() {
        return "Alumno{" +
                "nombre='" + nombre + '\'' +
                ", edad=" + edad +
                ", matricula='" + matricula + '\'' +
                ", notaMedia=" + notaMedia + // Este valor será 0.0 tras
deserialización
                '}';
    }
}
```

Y la clase para **guardar y leer** los objetos:

```
public class GuardarAlumno {
    public static void main(String[] args) {
        // Crear un objeto Alumno
        Alumno alumno = new Alumno("María López", 21, "A12345", 8.5);
        // Nombre del archivo donde se guardará el objeto
        String nombreArchivo = "alumno.dat";
        // Guardar el objeto Alumno en un archivo
        try (ObjectOutputStream oos = new ObjectOutputStream(new
```

```
FileOutputStream(nombreArchivo))) {

            oos.writeObject(alumno);

            System.out.println("Objeto Alumno guardado correctamente en el
archivo.");

        } catch (IOException e) {

            System.err.println("Ocurrió un error al guardar el objeto: " +
e.getMessage());

        }

        // Leer el objeto Alumno del archivo

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(nombreArchivo))) {

            Alumno alumnoLeido = (Alumno) ois.readObject();

            System.out.println("Objeto Alumno leído del archivo:");

            System.out.println(alumnoLeido);

        } catch (IOException | ClassNotFoundException e) {

            System.err.println("Ocurrió un error al leer el objeto: " +
e.getMessage());

        }

    }

}
```

Cambios realizados

- Campo `transient`:
  - Se añadió el campo `notaMedia` con el modificador `transient`.
  - Este campo no se serializa, por lo que al deserializar el objeto, su valor será el predeterminado (0.0 para double).
- Constructor actualizado:
  - Ahora el constructor recibe también el valor de `notaMedia`.
- Impresión del campo `notaMedia`:
  - Aunque el campo tiene un valor inicial (8.5), tras la deserialización aparecerá como 0.0.

Salida esperada

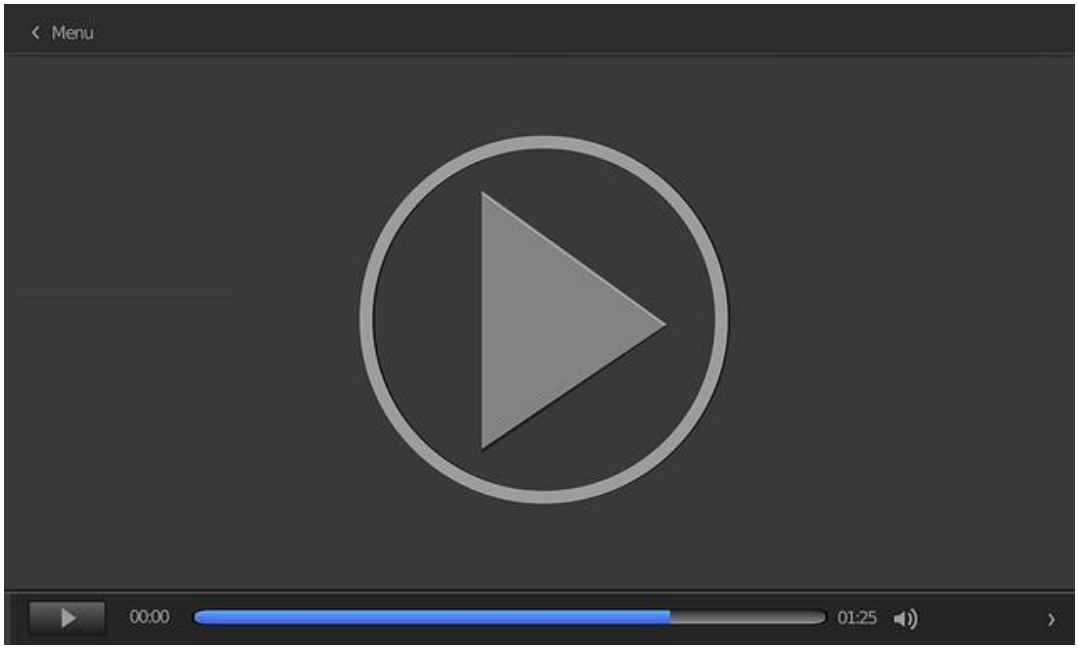
Al ejecutar este programa, la consola mostrará:

```
Objeto Alumno guardado correctamente en el archivo.

Objeto Alumno leído del archivo:

Alumno{nombre='María López', edad=21, matricula='A12345', notaMedia=0.0}
```

La clase File de Java

Adrian Lasso (2022, febrero 22). *Video 39 - La clase File de Java | Curso de POO en Java* [Video]. YouTube. https://www.youtube.com/watch?v=gfkiFBKHTSs

Este vídeo explora la clase `File` en Java. Detalla métodos comunes para obtener información sobre archivos y cómo implementar una aplicación que analiza documentos seleccionados por el usuario.



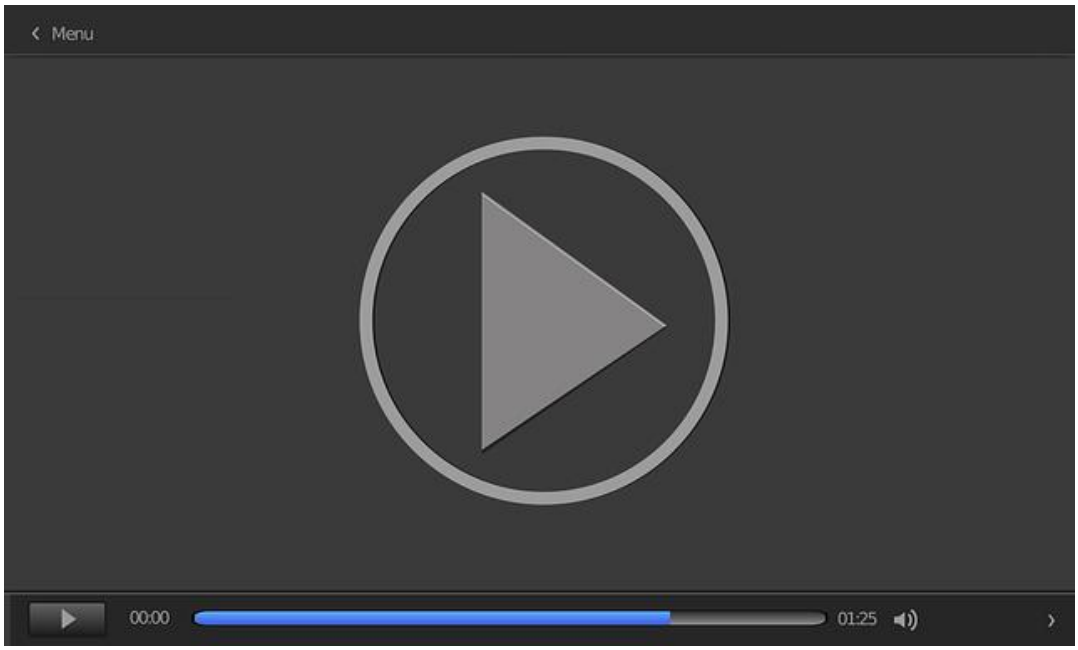
Accede al vídeo:

https://www.youtube.com/embed/gfkiFBKHTSs

Manejo de archivos con Java

UskoKruM2010 (2021). *Manejo de Archivos con Java (Clase File) | Curso Java #41* [Video]. YouTube. https://www.youtube.com/watch?v=NNh4cab-a5E

Aprende a manejar archivos en Java utilizando la clase `File`, con ejemplos prácticos que facilitan su comprensión y aplicación en proyectos reales.



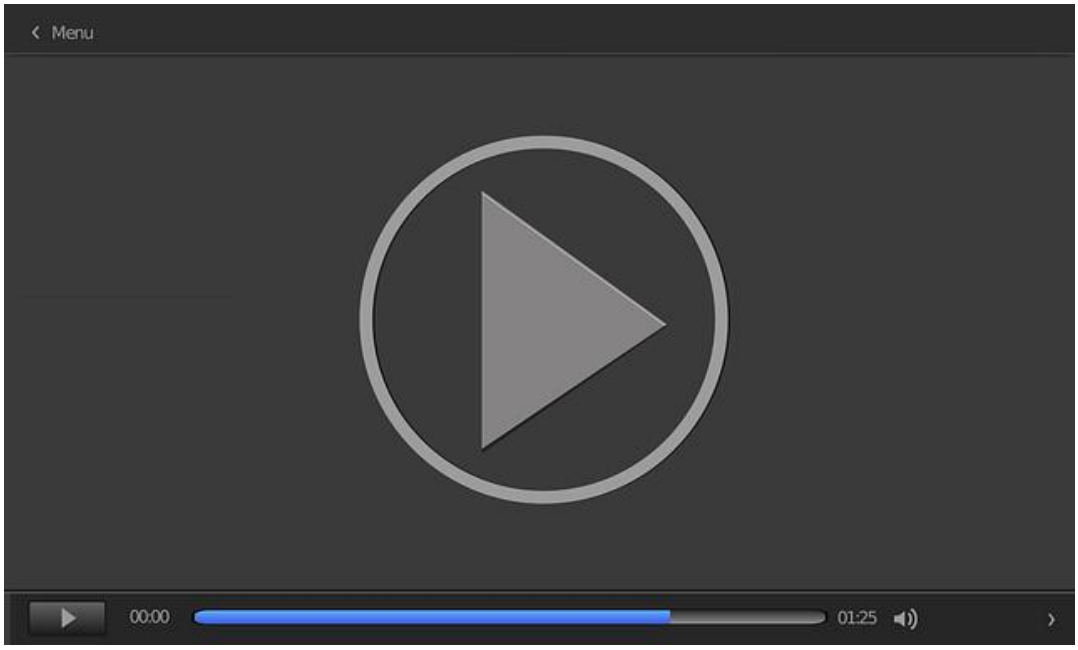
Accede al vídeo:

https://www.youtube.com/embed/NNh4cab-a5E

Como usar la clase File en Java

HackPress - Programación y Web (2023, septiembre 16). *Como USAR la clase FILE en JAVA #1 | ACCESO A DATOS* [Video]. YouTube. https://www.youtube.com/watch?v=8_FamWFTxX0

Este tutorial muestra cómo utilizar la clase `File` en Java, especialmente útil para estudiantes de Desarrollo de Aplicaciones Multiplataforma en la asignatura de Acceso a Datos.



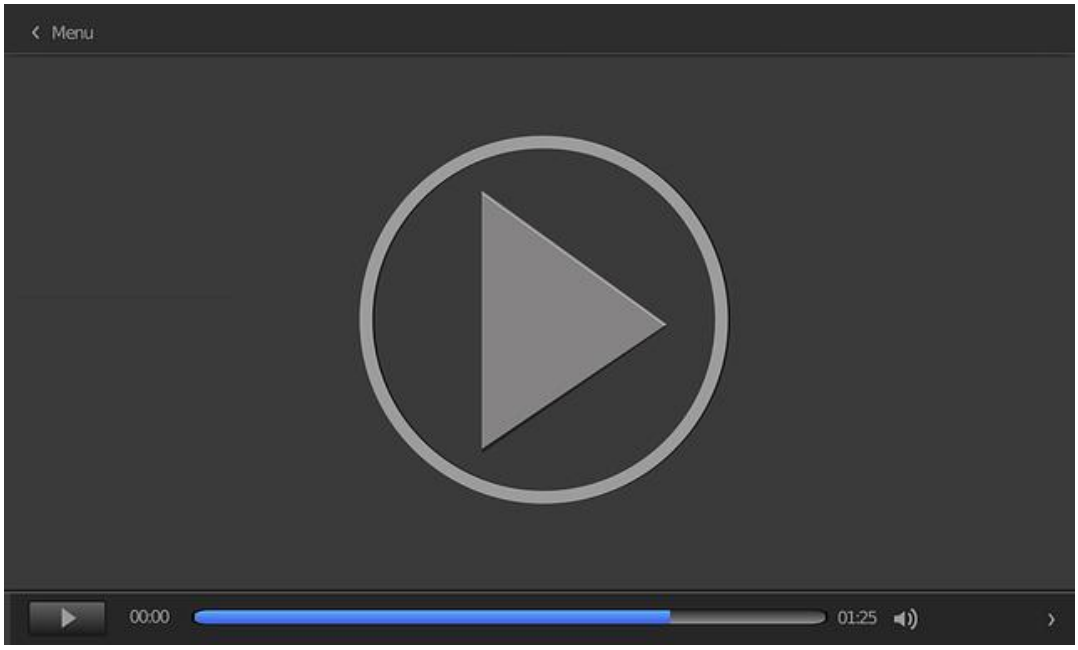
Accede al vídeo:

https://www.youtube.com/embed/8_FamWFTxX0

Ficheros de datos y ficheros de objetos

Programación y más (2021) *Ficheros en Java Segunda Parte: Ficheros de Datos y ficheros de objetos* [Video]. YouTube. https://www.youtube.com/watch?v=DZmrmLnn-0Y

Este vídeo aborda la creación, lectura y escritura de objetos en ficheros de datos y de objetos mediante Java. Profundiza en técnicas de persistencia.



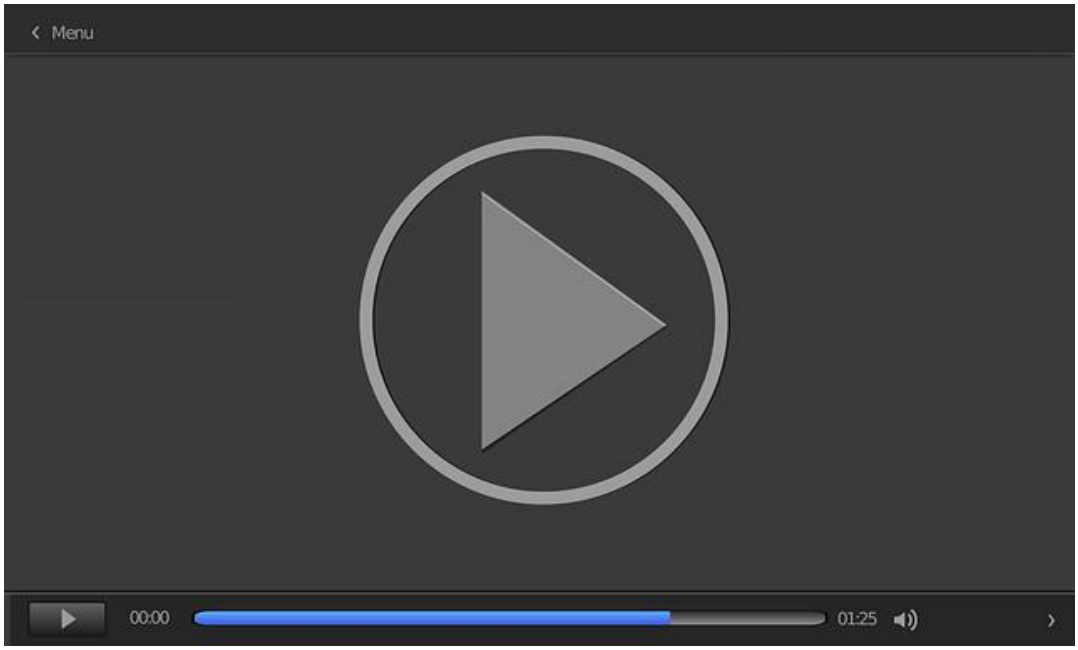
Accede al vídeo:

https://www.youtube.com/embed/DZmrmLnn-0Y

Serialización de objetos en Java

Aprende con Juan (2020) *Serialización de Objetos en Java* [Video]. YouTube. https://www.youtube.com/watch?v=YVOgtltq5FU

Presenta la serialización y persistencia de objetos en Java, detallando cómo convertir objetos en secuencias de *bytes* para su almacenamiento o transmisión.



Accede al vídeo:

https://www.youtube.com/embed/YVOgtltq5FU

Entrenamiento 1

Planteamiento del ejercicio

- **Crear un archivo:** escribe un programa en Java que cree un archivo llamado miArchivo.txt en el directorio actual. Si el archivo ya existe, el programa debe mostrar un mensaje indicando que el archivo ya estaba creado.

Desarrollo paso a paso

- 1. Importación de clases necesarias:
  - File: representa archivos y directorios.
  - IOException: excepción que puede lanzarse si ocurre un error de entrada/salida (por ejemplo, al intentar crear el archivo).

```
import java.io.File;

import java.io.IOException;
```

- 2. Creación del objeto File:
  - Se crea un objeto File que representa el archivo miArchivo.txt en el **directorio actual** del programa.
  - **Nota**: aún no se crea físicamente el archivo, solo se representa con un objeto.

```
File archivo = new File("miArchivo.txt");
```

- **3. Intento de creación del archivo.** createNewFile():
  - Intenta crear físicamente el archivo en el sistema.
  - Devuelve true si el archivo se crea correctamente.

- Devuelve false si el archivo ya existe.

```
if (archivo.createNewFile()) {
    System.out.println("Archivo creado: " + archivo.getName());
} else {
    System.out.println("El archivo ya existe.");
}
```

- **4. Manejo de errores:** si hay un error (por permisos, errores del sistema, etc.), se captura la excepción `IOException` y se muestra el mensaje de error.

```
} catch (IOException e) {
    System.err.println("Ocurrió un error al crear el archivo.");
    e.printStackTrace();
}
```

- **5. Resultado en consola:**

- Si el archivo no existía: archivo creado: `miArchivo.txt`
- Si el archivo ya existía: el archivo ya existe.

Solución

```
import java.io.File;

import java.io.IOException;

public class CrearArchivo {

    public static void main(String[] args) {

        File archivo = new File("miArchivo.txt");

        try {

            if (archivo.createNewFile()) {

                System.out.println("Archivo creado: " + archivo.getName());

            } else {

                System.out.println("El archivo ya existe.");

            }

        } catch (IOException e) {

            System.err.println("Ocurrió un error al crear el archivo.");

            e.printStackTrace();

        }

    }

}
```

Entrenamiento 2

Planteamiento del ejercicio

- **Listar el contenido de un directorio:** crea un programa que liste los nombres de los archivos y subdirectorios dentro de un directorio llamado `misArchivos` . Si el directorio no existe, el programa debe indicarlo.

Desarrollo paso a paso

- **1. Importación de la clase necesaria:** se importa la clase `File` , que permite representar y manipular archivos y directorios.

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

- **2. Creación del objeto `File` para el directorio:** se crea un objeto `File` que representa un directorio llamado `misArchivos` en el directorio actual.

```
File directorio = new File("misArchivos");
```

- **3. Verificar si el directorio existe y es un directorio:**

  - `exists()` : comprueba si el archivo o directorio existe en el sistema.
  - `isDirectory()` : comprueba si es un directorio (no un archivo).

```
if (directorio.exists() &&amp;amp; directorio.isDirectory()) {
```

- **4. Obtener la lista de archivos y subdirectorios:** devuelve un array de String con los **nombres** de todos los elementos dentro del directorio (`null` si hay un error al acceder o el directorio está vacío).

```
String[] contenido = directorio.list();
```

- **5. Mostrar el contenido si no es nulo:**

  - Si el contenido no es `null`, se recorren y muestran todos los nombres (archivos y subdirectorios).
  - Si es `null`, se muestra que está vacío (aunque técnicamente `list()` solo devuelve `null` si hay un error o no se puede acceder, no si está vacío).

```
System.out.println("Contenido del directorio:");
if (contenido != null) {
    for (String nombre : contenido) {
        System.out.println(nombre);
    }
} else {
    System.out.println("El directorio está vacío.");
}
```

- **6. Mensaje si el directorio no existe:** si no existe o no es un directorio, se informa al usuario.

```
} else {
    System.out.println("El directorio no existe.");
}
```

- Resultado posible:

  - Si el directorio existe y tiene archivos, se mostrará el contenido de este, por ejemplo: archivo1.txt , una subcarpeta o imagen.png .
  - Si el directorio no existe, se mostrará el mensaje «el directorio no existe».

Solución

```
import java.io.File;

public class ListarDirectorio {

    public static void main(String[] args) {

        File directorio = new File("misArchivos");

        if (directorio.exists() &&amp;amp; directorio.isDirectory()) {

            String[] contenido = directorio.list();

            System.out.println("Contenido del directorio:");

            if (contenido != null) {

                for (String nombre : contenido) {

                    System.out.println(nombre);

                }

            } else {

                System.out.println("El directorio está vacío.");

            }
```

```
        } else {

            System.out.println("El directorio no existe.");

        }

    }

}
```

Entrenamiento 3

Planteamiento del ejercicio

- **Leer un archivo línea por línea:** escribe un programa que lea el contenido de un archivo de texto llamado `datos.txt` y muestre solo aquellas líneas que contengan la palabra «hola» en la consola. Maneja las posibles excepciones si el archivo no existe.

Solución

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class LeerArchivo {

    public static void main(String[] args) {

        String nombreArchivo = "datos.txt";

        try (BufferedReader lector = new BufferedReader(new
FileReader(nombreArchivo))) {

            String linea;

            while ((linea = lector.readLine()) != null) {

            if (linea.contains("hola")){

                System.out.println(linea);

            }

            }
```

```
        } catch (IOException e) {

            System.err.println("Error al leer el archivo: " +
e.getMessage());

        }

    }

}
```

Entrenamiento 4

Planteamiento del ejercicio

- **Serializar y deserializar un objeto:** crea una clase `Persona` con los atributos `nombre` y `edad`. Serializa un objeto de esta clase en un archivo llamado `persona.dat` y luego léelo desde el archivo para mostrar los datos en la consola.

Desarrollo paso a paso

- **1. Importación de librerías:** se importan todas las clases necesarias para trabajar con archivos, flujos de entrada/salida y serialización:

  - `FileOutputStream`, `FileInputStream`: para escribir y leer archivos.
  - `ObjectOutputStream`, `ObjectInputStream`: para escribir y leer objetos.
  - `Serializable`: para que la clase Persona se pueda serializar.

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

- **2. Definición de la clase** `Persona`**:**

  - La clase **implementa** `Serializable`, lo cual permite convertir sus objetos a una secuencia de *bytes*.
  - Se define un campo `serialVersionUID` por seguridad (es opcional, pero recomendable).
  - Contiene los atributos `nombre` y `edad`, y un constructor para inicializarlos.

```
class Persona implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nombre;
```

```
    private int edad;

    ...

}
```

- **3. Sobrescribir** toString(): esto permite imprimir los datos del objeto de forma legible.

```
@Override

public String toString() {

    return "Persona{nombre='" + nombre + "', edad=" + edad + '}';

}
```

- **4. Crear una instancia de** Persona **:**
  - Se crea un objeto persona con datos concretos.
  - Se define el nombre del archivo donde se almacenará el objeto serializado.

```
Persona persona = new Persona("Juan Pérez", 30);

String archivo = "persona.dat";
```

- **5. Serialización del objeto:**
  - Se abre un ObjectOutputStream sobre un FileOutputStream .
  - writeObject **(persona)** escribe el objeto en el archivo persona.dat.
  - Se imprime un mensaje de éxito o error.

```
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(archivo))) {
```

```
    oos.writeObject(persona);

    System.out.println("Objeto serializado correctamente.");

} catch (IOException e) {

    ...

}
```

- **6. Deserialización del objeto:**
  - Se abre un `ObjectInputStream` para leer el archivo.
  - `readObject()` recupera el objeto serializado y se hace un *cast* a `Persona`.
  - Se imprime el contenido del objeto deserializado.
  - Se manejan posibles excepciones: errores de entrada/salida y clases no encontradas.

```
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(archivo))) {

    Persona personaLeida = (Persona) ois.readObject();

    System.out.println("Objeto deserializado: " + personaLeida);

} catch (IOException | ClassNotFoundException e) {

    ...

}
```

- **Resultado esperado en consola:**
  - Objeto serializado correctamente.

- Objeto deserializado: Persona{nombre='Juan Pérez', edad=30}

Solución

```
import java.io.*;

class Persona implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nombre;

    private int edad;

    public Persona(String nombre, int edad) {

        this.nombre = nombre;

        this.edad = edad;

    }

    @Override

    public String toString() {

        return "Persona{nombre='" + nombre + "', edad=" + edad + '}';

    }

}

public class SerializarPersona {

    public static void main(String[] args) {

        Persona persona = new Persona("Juan Pérez", 30);

        String archivo = "persona.dat";
```

```
        // Serializar

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(archivo))) {

            oos.writeObject(persona);

            System.out.println("Objeto serializado correctamente.");

        } catch (IOException e) {

            System.err.println("Error al serializar el objeto: " +
e.getMessage());

        }

        // Deserializar

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(archivo))) {

            Persona personaLeida = (Persona) ois.readObject();

            System.out.println("Objeto deserializado: " + personaLeida);

        } catch (IOException | ClassNotFoundException e) {

            System.err.println("Error al deserializar el objeto: " +
e.getMessage());

        }

    }

}
```

Entrenamiento 5

Planteamiento del ejercicio

- **Serialización de una lista de objetos con filtro:** crea una clase `Producto` con los atributos `nombre`, `precio` y `categoria`. Serializa una lista de productos en un archivo llamado `productos.dat`. Luego, deserializa los productos y muestra en la consola únicamente aquellos cuyo precio sea mayor a cincuenta.

Desarrollo paso a paso

- **1. Importaciones necesarias:** se importan clases para:
  - Entrada/salida de archivos y objetos (java.io.*)
  - Listas (`List` , `ArrayList`)
  - Filtrado con streams (`Collectors`)

```
import java.io.*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;
```

- **2. Definir la clase** `Producto`:
  - La clase **implementa** `Serializable` para que pueda guardarse en un archivo.
  - Tiene tres atributos: nombre, precio y categoría.

```
class Producto implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
    private String nombre;

    private double precio;

    private String categoria;

    ...

}
```

- **3. Constructor y método** getPrecio():
  - Se usa el constructor para crear los productos.
  - El método getPrecio() permite acceder al precio desde fuera de la clase, necesario para el filtro.

```
public Producto(String nombre, double precio, String categoria) {

    ...

}

public double getPrecio() {

    return precio;

}
```

- **4. Sobrescribir** toString(): permite mostrar los productos con un formato legible al imprimirlos en consola.

```
@Override

public String toString() {
```

```
    return "Producto{...}";

}
```

- **5. Crear una lista de productos:** se crea una lista de productos con distintos precios y categorías.

```
List<Producto> productos = List.of(

    new Producto("Producto A", 30, "Electrónica"),

    new Producto("Producto B", 60, "Hogar"),

    ...

);
```

- **6. Serializar la lista:** se usa un `ObjectOutputStream` para guardar la lista en el archivo productos.dat.

```
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(archivo))) {

    oos.writeObject(productos);

    System.out.println("Lista de productos serializada correctamente.");

}
```

- **7. Deserializar la lista:**
  - Se usa `ObjectInputStream` para leer la lista desde el archivo.
  - Se hace un cast a `List<Producto>`.

```
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(archivo))) {
```

```
    List<Producto> productosLeidos = (List<Producto>) ois.readObject();
    ...
}
```

- **8. Mostrar la lista deserializada:** se muestra la lista completa, incluyendo productos con precio menor o mayor a cincuenta.

```
System.out.println("Lista de productos deserializada:");
System.out.println(productosLeidos);
```

- **9. Filtrar productos con precio mayor a cincuenta:**
  - Se usa un stream para recorrer la lista y aplicar un filtro.
  - Se recogen los productos cuyo precio es mayor a cincuenta.

```
List<Producto> productosFiltrados = productosLeidos.stream()
        .filter(producto -> producto.getPrecio() > 50)
        .collect(Collectors.toList());
```

- **10. Mostrar productos filtrados:** se imprime solo la lista filtrada.

```
System.out.println("Productos con precio mayor a 50:");
productosFiltrados.forEach(System.out::println);
```

- **Ejemplo del resultado en consola:**
  - Lista de productos serializada correctamente.
  - Lista de productos deserializada:

```
[Producto{nombre='Producto A', precio=30.0, categoria='Electrónica'}, ...]
```

```
Productos con precio mayor a 50:

Producto{nombre='Producto B', precio=60.0, categoria='Hogar'}

Producto{nombre='Producto C', precio=100.0, categoria='Deportes'}

Producto{nombre='Producto E', precio=75.0, categoria='Moda'}
```

Solución

```
import java.io.*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;



// Clase Producto que implementa Serializable

class Producto implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nombre;

    private double precio;

    private String categoria;

    public Producto(String nombre, double precio, String categoria) {

        this.nombre = nombre;

        this.precio = precio;

        this.categoria = categoria;
```

```
    }

    public double getPrecio() {

        return precio;

    }

    @Override

    public String toString() {

        return "Producto{" +

                "nombre='" + nombre + '\'' +

                ", precio=" + precio +

                ", categoria='" + categoria + '\'' +

                '}';

    }

}

public class SerializarProductos {

    public static void main(String[] args) {

        String archivo = "productos.dat";

        // Crear lista de productos

        List<Producto> productos = List.of(

                new Producto("Producto A", 30, "Electrónica"),

                new Producto("Producto B", 60, "Hogar"),
```

```
                new Producto("Producto C", 100, "Deportes"),
                new Producto("Producto D", 25, "Papelería"),
                new Producto("Producto E", 75, "Moda")
        );
        // Serializar lista de productos
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(archivo))) {
            oos.writeObject(productos);
            System.out.println("Lista de productos serializada
correctamente.");
        } catch (IOException e) {
            System.err.println("Error al serializar la lista de productos:
" + e.getMessage());
        }
        // Deserializar lista de productos
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(archivo))) {
            List<Producto> productosLeidos = (List<Producto>)
ois.readObject();
            System.out.println("Lista de productos deserializada:");
            System.out.println(productosLeidos);
            // Filtrar productos con precio mayor a 50
            List<Producto> productosFiltrados = productosLeidos.stream()
```

```
                    .filter(producto -> producto.getPrecio() > 50)

                    .collect(Collectors.toList());

            System.out.println("Productos con precio mayor a 50:");

            productosFiltrados.forEach(System.out::println);

        } catch (IOException | ClassNotFoundException e) {

            System.err.println("Error al deserializar la lista de
productos: " + e.getMessage());

        }

    }

}
```