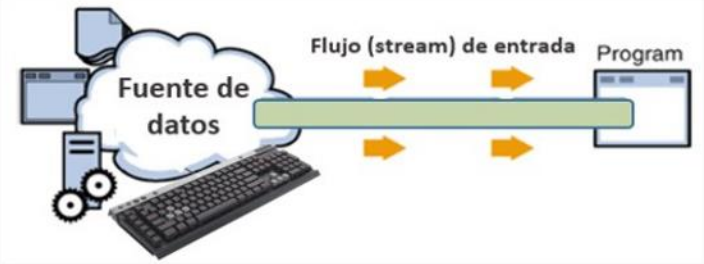
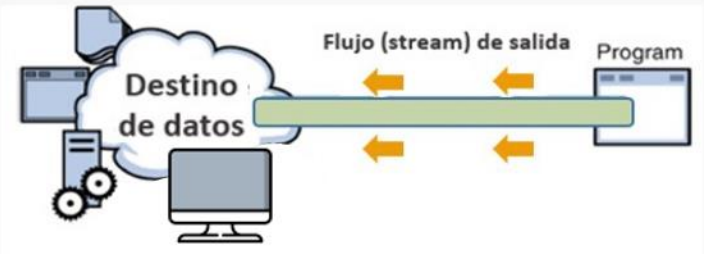
UT1. FICHEROS. PARTE 2

1. Flujos de entrada y salida en Java. Clases principales.

- Las aplicaciones muchas veces necesitan enviar datos a un determinado destino o leerlos de una determinada fuente
  - Ficheros en disco, red, memoria, otras aplicaciones, etc.
  - Esto es lo que se conoce como **E/S** (I/O en inglés)
- Esta E/S en Java se hace mediante **flujos (*streams*)**
  - Los datos se envían en serie a través del flujo
  - Se puede trabajar de la misma forma con todos los flujos, independientemente de su fuente o destino

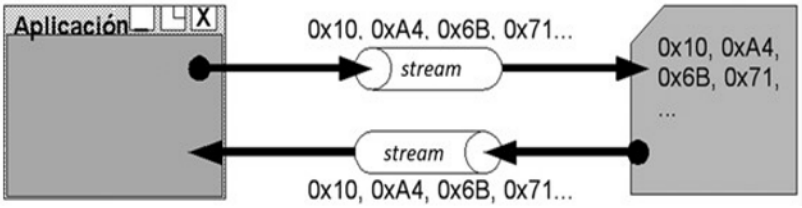


Flujos de entrada: Permiten a nuestro programa obtener datos desde un elemento externo (por ejemplo, el teclado o un fichero)



Flujos de salida: Permiten a nuestro programa mostrar los resultados del mismo en un elemento externo (por ejemplo, la consola/pantalla) o almacenarlos (escribirlos) en un fichero

- Importante: los datos se transmiten y procesan secuencialmente
  - **Flujos de lectura** : una vez leído un conjunto de datos, NO es posible volver atrás para leerlo de nuevo
  - **Flujos de escritura** : los datos se escriben, en el destino, exactamente en el mismo orden que se transmiten, no se pueden hacer saltos

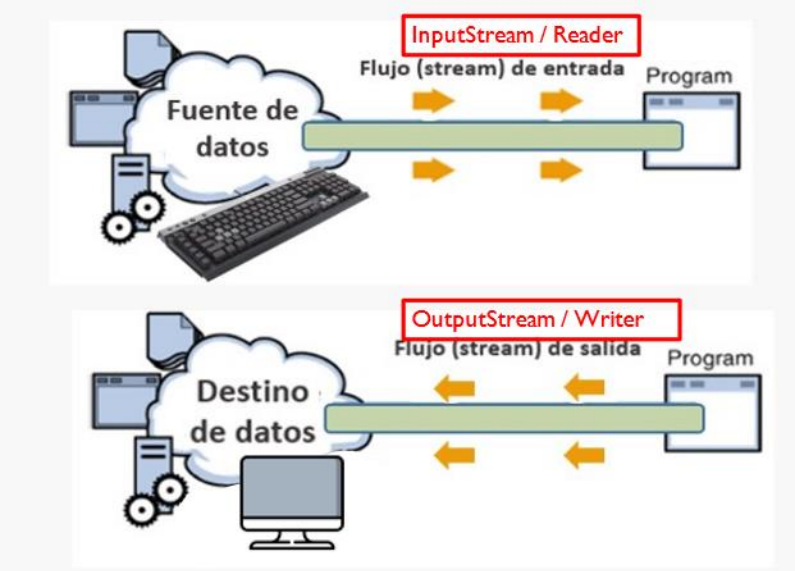


- Podríamos entender un flujo de datos como una tubería donde, en lugar de agua, se transmiten datos entre dos extremos → Una vez pasa el agua, ya no se puede echar atrás

Tipos de flujos según el tipo de datos

Según el tipo de datos que transportan, distinguimos:

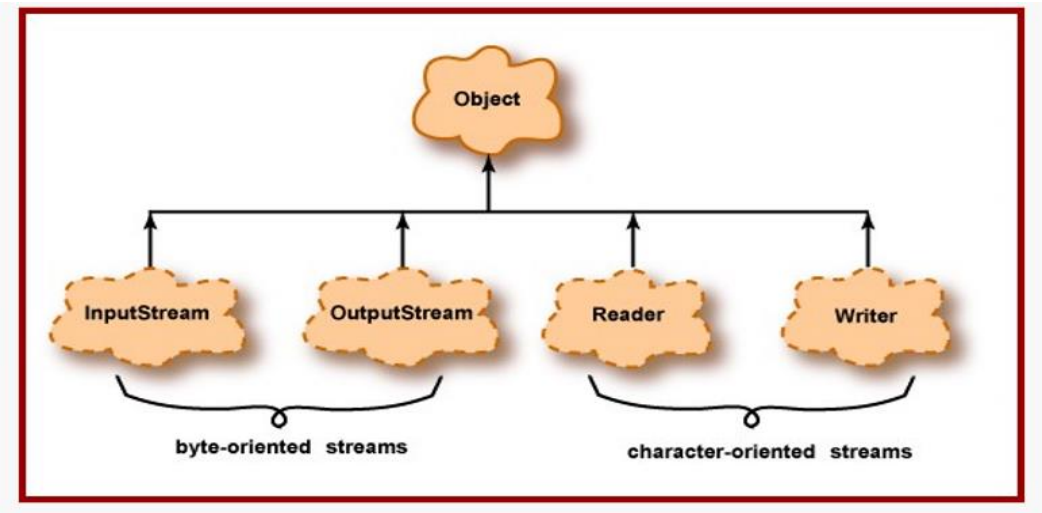
- **Flujos de bytes**, con sufijos InputStream prara flujos de entrada y OutputStream para flujos de salida
- **Flujos de caracteres**, con sufijos Reader para flujos de entrada y Writer para flujos de salida



Superclases

TIPO DE FLUJO	ENTRADA	SALIDA
BYTES	InputStream	OutpuStream
CARACTERES	Reader	Writer

Jerarquía simplificada de clases

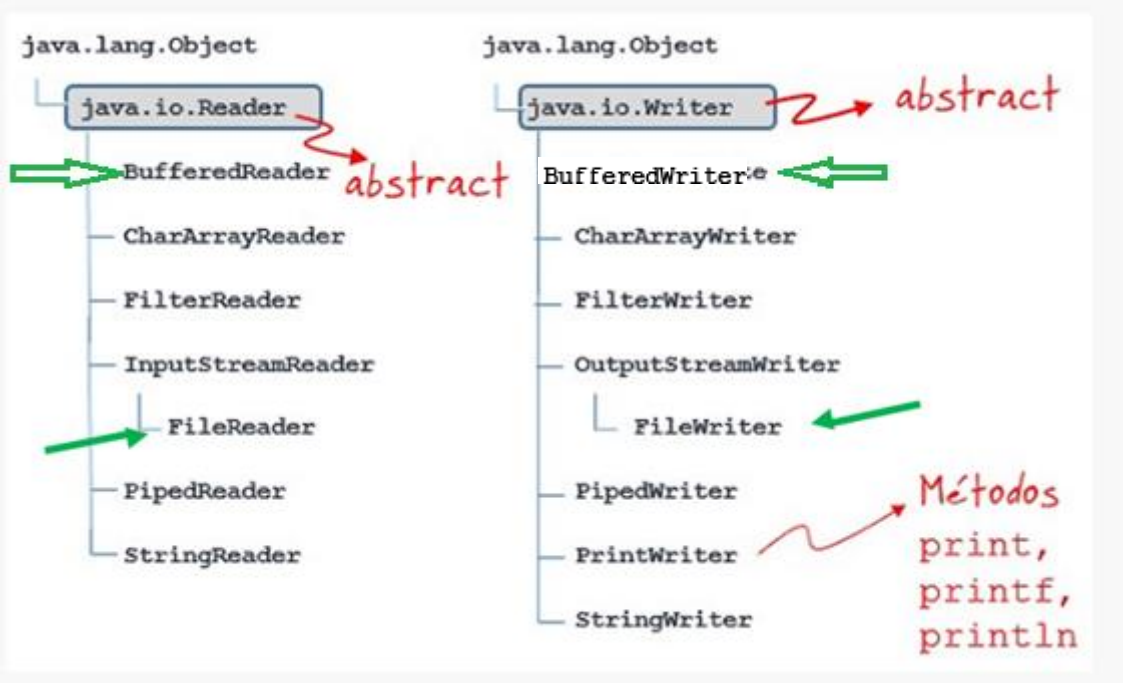


2. E/S en ficheros de texto: flujos de caracteres

Los flujos de caracteres en Java permiten operaciones como:

- Lectura de datos desde teclado
- Escritura de datos por pantalla
- Lectura/escritura de información desde/hacia un **fichero de texto** plano (.txt)

Clases Java para flujos de caracteres



- Las clases `BufferedReader` y `BufferedWriter` optimizan las operaciones de lectura y escritura al usar un buffer en lugar de acceder directamente al disco duro en cada operación.
- Con `FileReader` o `FileWriter`, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro.
    - Cuando se quiere leer o escriber un elevado número de caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.
- Las clases `BufferedReader` y `BufferedWriter` añaden un buffer intermedio entre nuestro programa Java y el archivo externo.
    - Cuando se lee o escribe, esta clase controla los accesos a disco.
    - Si vamos escribiendo, se guardarán los datos en el buffer hasta que haya bastantes datos como para hacer una lectura/escritura eficiente.

- **NOTA:** Las clases `FileWriter` y `FileReader` pueden usar un pequeño buffer implícito manejado por el sistema operativo o la JVM. Esto significa que, aunque no es un buffer controlado directamente por el programador, a nivel del sistema puede haber algún tipo de almacenamiento temporal en memoria (buffer de E/S), pero no es tan eficiente ni controlado como el buffer de las clases "Buffered" (ver métodos para lectura o escritura de arrays o strings en estas clases). Para tener un control y mejora en la eficiencia, las clases `BufferedWriter` y `BufferedReader` ofrecen buffering explícito, donde se almacena un bloque de datos en memoria antes de escribir o leer desde el archivo.

Flujos de entrada de texto

- Los flujos de **entrada** de tipo texto heredan de la clase `Reader`.
- Las clases de entrada de texto tienen siempre un nombre que termina en `Reader`.
- Para leer desde ficheros de texto usaremos las clases `FileReader (hereda de InputStreamReader,` que a su vez hereda de `Reader) y BufferedReader` (hereda directamente de `Reader`).

La Clase FileReader

- Permite la escritura de **caracteres individuales**.
- Constructores principales de la clase `FileReader`:
    - `FileReader(File file)`
    - `FileReader(String fileName)`

      Donde `fileName` puede incluir la ruta de `acceso al fichero`
- Los constructores de FileReader pueden arrojar una excepción del tipo `IOException`,
- Ejemplo:

```
FileReader in = new FileReader ("/home/alumno/prueba.txt");
```

Crea un flujo de texto asociado al archivo `/home/alumno/prueba.txt`

La Clase BufferedReader

- Permite la lectura eficiente de **caracteres individuales, arrays y cadenas**.
- Se pueden leer líneas completas.
- Constructores principales de la clase `BufferedReader` (ambos reciben un objeto Reader o de clases que heredan de reader, como FileReader):
    - `BufferedReader (Reader in):` Crea un flujo de entrada de caracteres con un buffer de tamaño predeterminado.
    - `BufferedReader (Reader in, int size):` Crea un flujo de entrada de caracteres con un buffer de tamaño sz

Flujos de salida de texto

- Los flujos de **salida** de tipo texto heredan de la clase Reader.
- Las clases de salida de texto tienen siempre un nombre que termina en `Writer`.
- Para escribir en ficheros de texto usaremos las clases `FileWriter (hereda de OutputStreamWriter,` que a su vez hereda de la `Writer`) `y BufferedWriter` (hereda directamente de `Writer`).

La Clase FileWriter

- Permite la escritura de **caracteres individuales**.
- Constructores principales de la clase `FileWriter`:
    - `FileWriter`(File file)
    - `FileWriter(String fileName)`
    - `FileWriter`(File file, boolean append)
    - `FileWriter(String fileName, boolean append)`

Los dos primeros constructores destruyen la versión anterior del archivo y escriben en él desde el principio. En los dos últimos constructores, cuando el parámetro booleano `append` vale `true`, el texto se añade al final del archivo, respetando el contenido anterior.

Si el archivo no existe lo crea, siempre que sea posible.

- Los constructores de FileWriter pueden arrojar una excepción del `IOException`.

La Clase BufferedWriter

- Permite la escritura eficiente de **caracteres individuales, arrays y cadenas**.
- Constructores principales de la clase `BufferedWriter` (ambos reciben un objeto Writer o de clases que heredan de Writer, como File Writer):
  - `BufferedWriter(Writer out)`: crea un flujo de salida de caracteres con un buffer de tamaño predeterminado.
  - `BufferedWriter(Writer out, int sz)`: crea un flujo de salida de caracteres con un buffer de tamaño sz.

Flujo sin *buffering* y con *buffering*:

Flujo sin buffering	Flujo con buffering
`new FileReader("f.txt")`	`new BufferedReader(new FileReader("f.txt"))`
`new FileWriter("f.txt")`	`new BufferedWriter(new FileWriter("f.txt"))`

Métodos para flujos de caracteres

Métodos para flujos tipo Reader		
Clase	**Método**	**Descripción**
`FileReader` `BufferedReader`	`int read()`	Lee el siguiente carácter desde el Reader y devuelve un carácter Unicode codificado como entero (si queremos recuperar el carácter asociado debemos aplicarle un cast). Devuelve -1 si no se pueden leer más caracteres. Puede lanzar una excepción del tipo **IOException**.
`FileReader` `BufferedReader`	`public int read(char[] cbuf)`	Lee caracteres y los guarda en el array `cbuf`. Devuelve el número de caracteres leídos, o -1 si se llega al final del flujo. Puede lanzar una excepción del tipo **IOException**.
`FileReader` `BufferedReader`	`public int read(char[] cbuf, int offset, int length)`	Lee un máximo de `length` caracteres y los guarda en el array `cbuf`, a partir de la posición `offset`. Devuelve el número de caracteres leídos, o -1 si se llega al final del flujo. Puede lanzar una excepción del tipo **IOException**.

Métodos para flujos tipo Reader		
Clase	**Método**	**Descripción**
`BufferedReader`	`public String readLine()`	Lee una línea completa del archivo de texto hasta el salto de linea '\n' o retorno de carro '\r', sin incluir estos caracteres en la cadena devuelta. Al llegar al final del fichero devuelve `null`. Puede lanzar una excepción del tipo **IOException**.
`FileReader` `BufferedReader`	`int skip(long n)`	Hace que se omitan los n primeros caracteres. Devuelve los caracteres que realmente se han omitido. Puede lanzar una excepción del tipo **IOException**.
`FileReader` `BufferedReader`	`void close()`	Cierra el Reader. Puede lanzar una excepción del tipo **IOException**.

Métodos para flujos tipo Writer		
Clase	**Método**	**Descripción**
`FileWriter` `BufferedWriter`	`void write(int c)`	Escribe un carácter (los últimos 16 bits del int pasado como parámetro) en el Writer. Puede lanzar una excepción del tipo **IOException**.
`FileWriter` `BufferedWriter`	`void write(char[] cbuf)`	Escribe el array `cbuf`. Puede lanzar una excepción del tipo **IOException**.

`FileWriter`
`BufferedWriter`
	`void write(char[] cbuf,`
	`int offset,`
	`int length)`
		Escribe un total de `length` caracteres de un array `cbuf`, a partir de la posición `offset`. Puede lanzar una excepción del tipo **IOException**.

`FileWriter` `BufferedWriter`	`void write(String cadena)`	Escribe una cadena de caracteres en el Writer. Puede lanzar una excepción del tipo **IOException**.

`FileWriter`
`BufferedWriter`
	`void write(String cadena,`
	`int offset,`
	`int length)`
		Escribe un total de `length` caracteres de un String cadena, a partir de la posición `offset`. Puede lanzar una excepción del tipo **IOException**.

`FileWriter` `BufferedWriter`	`void flush()`	Vacía el flujo de salida actual del Writer y fuerza la escritura de los datos en el fichero. Puede lanzar una excepción del tipo **IOException**.
`FileWriter` `BufferedWriter`	`void close()`	Cierra el flujo Writer, vaciándolo antes. Puede lanzar una excepción del tipo **IOException**.
`BufferedWriter`	`void newLine()`	Escribe un separador de línea. El separador de línea está definido en la propiedad del sistema `line.separator`, y no se corresponde necesariamente con el carácter '\n'.

Ejemplos con clases FileReader y FileWriter

Ejemplo 01. Clase FileReader: Leer el contenido de un fichero de texto carácter a carácter y mostrarlo por pantalla

```
/*****************************************************************************
/*  Clase LeerFichTexto: operaciones de lectura de ficheros
/*  de texto con FileReader
/*****************************************************************************/
class LeerFichTexto{

    //-------------------------------------------------------------
    //Método lee(): pide un nombre de un archivo de texto y lo lee
    //carácter a carácter, mostrando el contenido en pantalla
    //-------------------------------------------------------------
    public void lee(){//Versión capturando las posibles IOException

        System.out.println("Introduce el fichero que quieres leer: ");
        Scanner sc = new Scanner(System.in);
        File fich = new File(sc.nextLine()); //Crea el objeto File que referencia al fichero

        try{
            FileReader entrada = new FileReader (fich); //Crea el stream "entrada" de tipo FileReader
                                                        //Puede lanzar IOException
            int car=entrada.read(); //Lee el primer carácter del stream "entrada".
                                    //También puede lanzar IOException

            //Mientras no se llegue al final del fichero imprime el carácter en pantalla
            //y lee el siguiente carácter del fichero, como entero
            while (car!=-1){
                System.out.print((char)car);//Muestra el carácter leído convertido a char
                car=entrada.read();
            }
            entrada.close();//Cerramos el stream. Puede lanzar IOException

        } catch (IOException ioe){
            System.out.println("ERROR EN METODO Lee: " + ioe);
        }
    }  //Fin método lee()
}  //Fin clase LeerFichero
```

Ejemplos de ejecución:

```
Introduce el fichero que quieres leer:
/home/alumno/Documentos/hola.txt
Hola.
Este es un archivo de texto.
Hasta pronto.
BUILD SUCCESSFUL (total time: 13 seconds)
```

```
run:
Introduce el fichero que quieres leer:
/home/alumno/hola
ERROR EN METODO Lee: java.io.FileNotFoundException: /home/alumno/hola (No existe el archivo o el directorio)
BUILD SUCCESSFUL (total time: 10 seconds)
```

Ejemplo 02. Clase FileWriter: Escribir en un fichero de texto carácter a carácter

Nota: opciones para acceder a cada carácter de un String. Sea la variable

```
String cadena;
```

a) Convertir el `String` a un array de `char` usando el método **`toCharArray()`** de la clase `String`:

```
Char[] arrayDeChar = cadena.toCharArray();
```

y ya podemos acceder a cada posición del arrayDeChar y obtener el carácter correspondiente, por ejemplo, a la posición 3:

```
char c = arrayDeChar[3];
```

b) Utilizar el método **`charAt(int i)`** de la clase `String` para acceder al carácter en la posición i de la cadena. Por ejemplo, para acceder a la posición 3:

```
char c = cadena.charAt(3);
```

```java
/* Clase EscribirFichTexto: operaciones de escritura en ficheros
* de texto con FileWriter
*****************************************************************************/
class EscribirFichTexto{
    //-------------------------------------------------------------
    //Método escribe(): pide una cadena por consola y la escribe carácter
    //a carácter en un archivo cuyo nombre también se pide por consola
    //-------------------------------------------------------------
    public void escribe(){
        try {

            System.out.println("Nombre del fichero de texto que quieres escribir:");
            Scanner sc = new Scanner(System.in);
            File fich= new File(sc.nextLine());//Objeto File que encapsula al fichero

            //Creamos el stream "salida" tipo FileWriter. Si el archivo no existe, lo crea.
            FileWriter salida = new FileWriter(fich);//Si el archivo existe, lo sobreesribe
                                                     //Puede lanzar IOException
            //Leemos la frase que hay que escribir en el fichero
            System.out.println("Frase que quieres escribir en el fichero " + fich.getName() + ":");
            String frase = sc.nextLine();

            //Escribimos la frase en el fichero carácter a carácter, usando el método charAt(int i) de String
            //Empezamos en la posicón 0, hasta length -1
            for (int i=0;i<frase.length(); i++){
                salida.write((int) frase.charAt(i));
            }   //Cerramos el stream FileWriter

            salida.close(); //Cierra el stream "salida" de tipo FileWriter. Puede lanzar IOException
        } //Fin escribe()
        catch (IOException ioe) {
            System.out.println("ERROR EN METODO Escribe: " + ioe);
        }
    }
}//Fin clase EscribirFichTexto
```

Ejemplo 03. Clase BufferedWriter: Escribir en un fichero de texto línea a línea

```java
package ut1_24_25_ejemplos;

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

public class UT1_24_25_EjemploBufferedWriter {
    public static void main(String[] args) {
        File ruta = new File(System.getProperty("user.dir")
                + "/src/fichero.txt");
        System.out.println("Archivo a escribir:" + ruta.getAbsolutePath());//Muestra ruta y nombre del fichero
        BufferedWriter bfw = null;

        try {
            bfw = new BufferedWriter(new FileWriter(ruta));
            for (int i = 1; i <= 5; i++) {
                bfw.write("Fila número: " + i);
                bfw.newLine();
            }
        } catch (IOException ex) {
            System.out.println("ERROR en UT1_24_25_EjemploBufferedWriter: " + ex);
        } finally {
            if (bfw != null) {
               try {
                    bfw.close();
               } catch (IOException ex) {
                    System.out.println("ERROR en UT1_24_25_EjemploBufferedWriter: " + ex);
               }//Fin try-catch
            }//Fin if
        }//Fin try-catch-finally
    }//Fin main
}//Fin UT1_24_25_EjemploBufferedWriter
```

Ejemplo 04. Clase BufferedReader: Leer de un fichero de texto línea a línea

Tres versiones que varían en el tratamiento de las excepciones.

<u>Versión 1</u>: usa "throws" para propagar las excepciones hacia arriba

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

/*****************************************************************************
 * Lee un fichero de texto línea a línea y muestra el contenido por pantalla
 * Usa throws para propagar las excepciones hacia arriba
 * No hay que capturarlas con try-catch
 *****************************************************************************/

public class UT1_24_25_EjemploBufferedReader {
    public static void main(String[] args) throws IOException, FileNotFoundException{
        File ruta = new File(System.getProperty("user.dir")
                + "/src/fichero.txt"); //Crea objeto File con la ruta dentro del directorio actual
        System.out.println("Contenido del archivo " + ruta.getAbsolutePath() + ":\n");//Muestra ruta y nombre del fichero

        BufferedReader bfr = new BufferedReader(new FileReader(ruta));//Abre stream BufferedReader
        String linea;
        while ((linea = bfr.readLine()) != null) { //Lee del stream línea a línea
            System.out.println(linea);  //Muestra en pantalla la línea leída
        }//Fin while
        bfr.close(); //Cierra el stream
    }//Fin main
}//Fin UT1_24_25_EjemploBufferedReader
```

Versión 2: Usa try-catch-finally, con un try-catch anidado para el cierre del stream

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

/******************************************************************************
 * Lee un fichero de texto línea a línea y muestra el contenido por pantalla
 * Usa varios bloques try-catch-finally y un bloque try-catch anidados
 ******************************************************************************/
public class UT1_24_25_EjemploBufferedReader {
    public static void main(String[] args) {
        File ruta = new File(System.getProperty("user.dir")
                + "/src/fichero.txt"); //Crea objeto File con la ruta dentro del directorio actual
        System.out.println("Contenido del archivo " + ruta.getAbsolutePath() + ":\n");//Muestra ruta y nombre del fichero
        BufferedReader bfr = null;

        try {
            bfr = new BufferedReader(new FileReader(ruta)); //Abre stream BufferedReader
            String linea;
            while ((linea = bfr.readLine()) != null) { //Lee del stream línea a línea
                System.out.println(linea);      //Muestra en pantalla la línea leída
            }
        } catch (FileNotFoundException ex) {
            System.out.println("ERROR en UT1_24_25_EjemploBufferedReader. No se encuentra el archivo: " + ex);
        } catch (IOException ex) {
            System.out.println("ERROR en UT1_24_25_EjemploBufferedReader: " + ex);
        } finally {
            if (bfr != null) { //Si se abrió el stream
                try {
                    bfr.close(); //Cierra el stream
                } catch (IOException ex) {
                  System.out.println("ERROR en UT1_24_25_EjemploBufferedReader: " + ex);
                }//Fin try-catch
            }//Fin if
        }//Fin try-catch-finally
    }//Fin main
}//Fin UT1_24_25_EjemploBufferedReader
```

Versión 3: Usa try-with-resources

```
package ut1_24_25_ejemplos;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;


/******************************************************************************
 * Lee un fichero de texto línea a línea y muestra el contenido por pantalla
 * Usa try-with-resources
 ******************************************************************************/
public class UT1_24_25_EjemploBufferedReader {
    public static void main(String[] args) {
        File ruta = new File(System.getProperty("user.dir")
                + "/src/fichero.txt"); //Crea objeto File con la ruta dentro del directorio actual
        System.out.println("Contenido del archivo " + ruta.getAbsolutePath() + ":\n");//Muestra ruta y nombre del fichero

        try(BufferedReader bfr = new BufferedReader(new FileReader(ruta))){//Abre stream BufferedReader
            String linea;
            while ((linea = bfr.readLine()) != null) { //Lee del stream línea a línea
                System.out.println(linea);      //Muestra en pantalla la línea leída
            }
        } catch (FileNotFoundException ex) {
            System.out.println("ERROR en UT1_24_25_EjemploBufferedReader. No se encuentra el archivo: " + ex);
        } catch (IOException ex) {
            System.out.println("ERROR en UT1_24_25_EjemploBufferedReader: " + ex);
        }
    }//Fin main
}//Fin UT1_24_25_EjemploBufferedReader
```

Ejemplo de ejecución:

```
run:
Contenido del archivo /home/alumno/NetBeansProjects/UT1_24_25/UT1_24_25_Ejemplos/src/fichero.txt:

Fila número: 1
Fila número: 2
Fila número: 3
Fila número: 4
Fila número: 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

3. La clase Scanner y flujos de entrada desde archivos de texto

En Java, la clase Scanner se utiliza principalmente para leer y analizar secuencias de caracteres. Aunque generalmente se usa para leer desde la consola (System.in), también puede emplearse con tipos de flujos de texto, como un BufferedReader, para leer desde archivos de texto. Lo que hace la clase Scanner es acceder al buffer en busca de secuencias de caracteres (tokens) que coincidan con el tipo de datos requerido por el método invocado: nextInt(), next(), etc. Por tanto, Scanner, además de leer secuencias de caracteres es capaz de analizarlas y convertirlas en tipos de datos primitivos. Por defecto, los tokens deben estar separados por caracteres blancos (espacios, tabuladores y cambios de línea), o secuencias de ellos.

Usar un Scanner con un flujos de tipo texto permite leer archivos de texto línea por línea, palabra por palabra o incluso analizar valores numéricos de una manera más sencilla.

Podemos usar diferentes constructores de la clase Scanner para leer de archivos de texto, entre otros:

- Pasar al constructor del Scanner un objeto File. El Scanner se encarga internamente de crear el flujo de entrada desde el archivo. Por ejemplo:

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

Scanner sc = new Scanner(file);
```

- BufferedReader y Scanner: envolver un FileReader dentro de un BufferedReader y luego pasarlo al Scanner. Por ejemplo:

```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));

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

Ejemplo 05. Uso de Scanner con BufferedReader

```java
/**************************************************************
 * Lee un archivo de texto mediante un Scanner asociado a un
 * BufferedReader y muestra su contenido en pantalla
 **************************************************************/

public class UT1_24_24_EjemploScannerYBufferedReader {

    public static void main(String[] args) {
        String rutaArchivo = "archivo.txt"; // Ruta del archivo que se desea leer

        // Uso de try-with-resources para manejar recursos automáticamente
        try (BufferedReader br = new BufferedReader(new FileReader(rutaArchivo));
             Scanner scanner = new Scanner(br)) {

            // Leer el archivo usando Scanner y va mostrando las líenas leídas
            int i=1;
            while (scanner.hasNextLine()) {
                String linea = scanner.nextLine();
                System.out.println("Línea" + i + ": " + linea);
                i++;
            }
        } catch (IOException e) {
            System.err.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

➢ Ejercicios 8 y 9 de la hoja de ejercicios UT1 Parte 2

4. Escritura con formato en ficheros de texto: clase PrintWriter

La clase PrintWriter en Java se utiliza para escribir datos de texto en archivos con formato. Aporta métodos similares a los de System.out: print(), println() y printf().

Constructores

- `PrintWriter(String fileName)`
- `PrintWriter(File file)`
- PrintWriter(OutputStream out)

Métodos principales para escritura

- `void print(...)`: Imprime los datos que se le pasan (puede ser un número, cadena, objeto, etc.) sin un salto de línea.
- `void println(...)`: Imprime los datos y añade un salto de línea después.

- `void printf (String format, Object... args)`: Imprime los datos especificados a partir del segundo argumento con el formato indicado en el primer argumento (como printf en C).
- `void flush()`: Fuerza a que todos los datos en el buffer se escriban en el archivo.
- `void close()`: Cierra el PrintWriter y libera los recursos.

Ejemplo 06. Escritura con PrintWriter

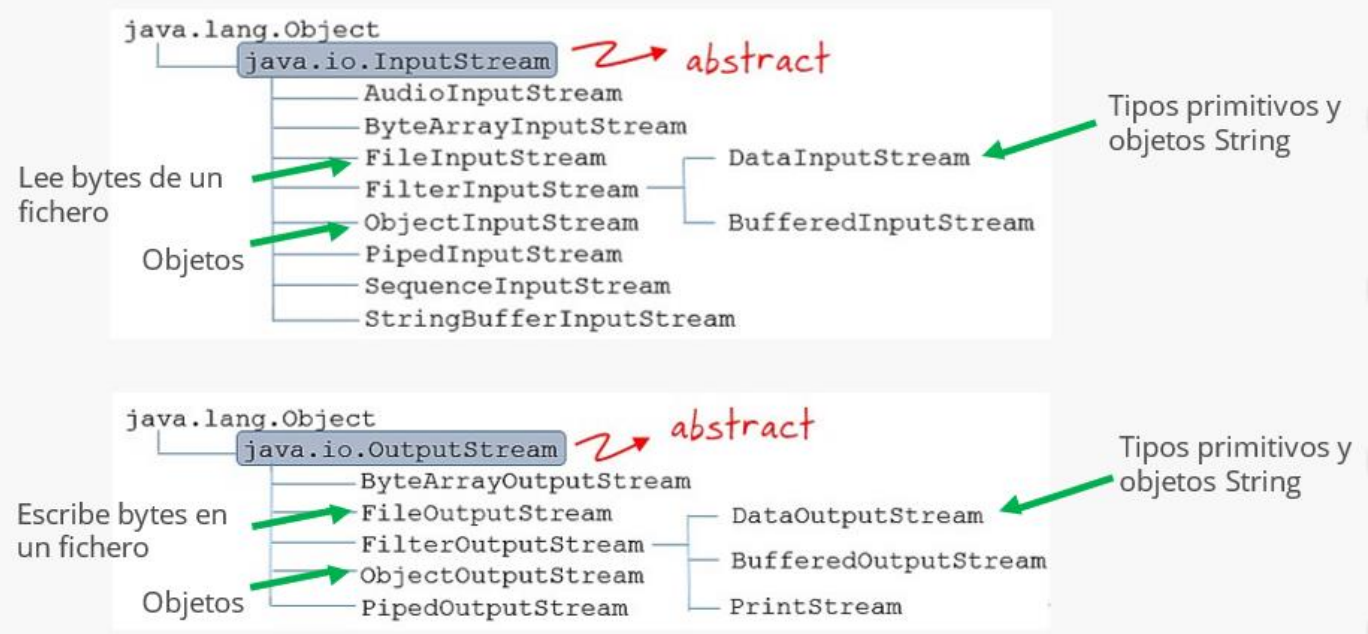
```java
/* ************************************************************************
 * Escribe texto en un archivo usando diferentes métodos de la clase PrintWriter
 ************************************************************************/
public class UT1_24_25_EjemploPrintWriter {

    public static void main(String[] args) {
        String rutaArchivo = "salida.txt";

        // Crear PrintWriter dentro de un bloque try-with-resources
        // Se cerrará el PrintWriter de forma automática
        try (PrintWriter writer = new PrintWriter(new FileWriter(rutaArchivo))) {
            writer.println("Hola, Mundo!"); // Escribe una línea en el archivo
            writer.print("Este es un ejemplo de "); // Escribe sin salto de línea
            writer.println("PrintWriter."); // Ecribe con salto de línea

            // Formatear salida
            writer.printf("Número: %.2f%n", 3.14159); // Imprime el número formateado con dos decimales
        } catch (IOException ioe) {
            System.out.println(ioe); // Imprime las excepciones de entrada/salida
        }
    }
}
```

➢ Ejercicios 10, 11 y 12 de la hoja de ejercicios UT1 Parte 2

5. E/S en ficheros binarios: flujos de bytes

Los flujos de bytes en Java proporcionan las herramientas necesarias para la gestión de entradas y salidas de bytes. Su uso está orientado a la lectura y escritura de datos binarios.

Clases Java para flujos de bytes



- Todos los flujos de bytes en Java heredan de las superclases abstractas `InputStream` y `OutputStream`. Estas especifican los métodos relativos al comportamiento común a cualquier flujo de bytes, y cada subclase se encarga de sobreescribirlos o añadir otros nuevos, según sus particularidades.
- **`InputStream / OutputStream`:** Clases **abstractas** para trabajar con flujos binarios de datos (lectura y escritura de bytes).
- **`FileInputStream / FileOutputStream`:** Implementaciones de las anteriores para operaciones de lectura/escritura en archivos binarios.
- **`FilterInputStream / FilterOutputStream`:** clases **abstractas** que permiten a las clases que heredan de ellas agregar funcionalidades adicionales al procesamiento de los datos. Clases que heredan de estas:
    - **`DataInputStream / DataOutputStream`:** permiten la lectura/escritura de datos primitivos (int, float, etc.).
    - **`BufferedInputStream / BufferedOutputStream`:** añaden un buffer para mejorar la eficiencia en la escritura de datos.
    - **`PrintStream`** `(hereda de FilterOutputStream)`: facilita la impresión de valores formateados.
- **`ObjectInputStream / ObjectOutputStream`:** Permiten trabajar con objetos (serialización y deserialización).
- Los constructores de estas clases son similares a los de FileReader/FileWriter

Comparación y casos de uso de las clases principales

Lectura

Clase	Descripción	Casos de uso
`FileInputStream`	Leer bytes sin interpretación, datos crudos.	Archivos binarios como imágenes o videos, flujos de bytes sin estructura definida.
`DataInputStream`	Leer datos primitivos (int, float, boolean) en formato binario.	Archivos que contienen datos primitivos escritos de manera binaria, lectura estructurada de tipos primitivos.
`ObjectInputStream`	Leer objetos serializados desde un archivo.	Archivos que contienen objetos serializados, como listas, mapas, o instancias de clases personalizadas.

Escritura

Clase	Descripción	Casos de uso
`FileOutputStream`	Escribir bytes sin interpretación (datos crudos).	Guardar datos binarios en un archivo, como imágenes, vídeos, archivos de audio o cualquier dato en formato sin estructura específica.
`DataOutputStream`	Permite escribir datos primitivos (int, char, etc.) en un formato binario estructurado.	Escribir datos primitivos en un archivo, por ejemplo, cuando se necesita conservar un formato de datos específico.
`ObjectOutputStream`	Escribe objetos en un archivo en formato binario mediante serialización, para poder ser leídos y reconstruidos más tarde.	Serialización de objetos: guardar el estado de un objeto en un archivo para su posterior recuperación.

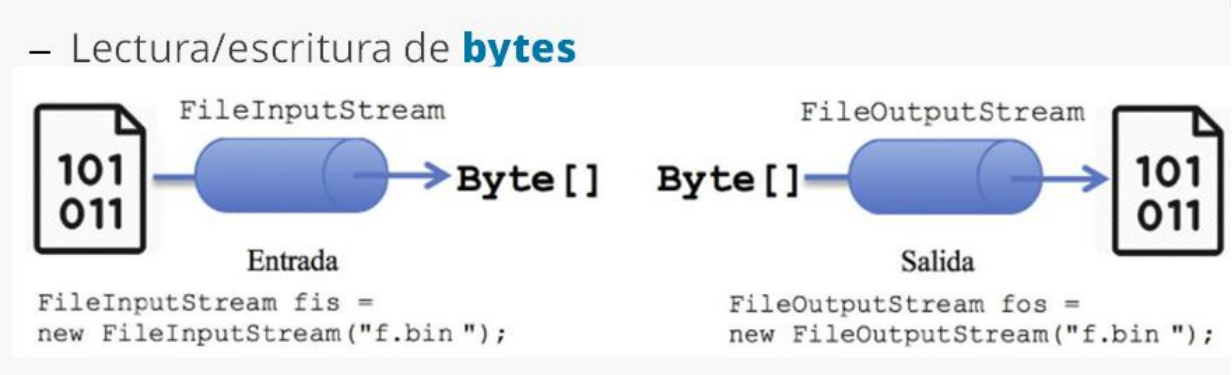
Métodos para flujos de bytes

Métodos para lectura: InputStream	
Método	**Descripción**
`int available()`	Devuelve el número de bytes que se pueden leer del InputStream sin que se produzca bloqueo. Puede lanzar una excepción del tipo **IOException**.
`int read()`	Lee el siguiente byte desde el InputStream. Devuelve -1 si no se pueden leer más byes. El valor del byte leído se devuelve como un entero en el rango 0 a 255.

Métodos para lectura: InputStream	
Método	**Descripción**
`int read(byte[] b)`	Lee bytes y los guarda en el array `b`. Intenta leer hasta `b.length` bytes. Devuelve el número de bytes leídos, o -1 si se llega al final del flujo. Puede lanzar una excepción del tipo **IOException.**
`int read(byte[] b, int off, int len)`	Lee hasta un total de `len` bytes del stream en un array de bytes y los guarda en el array b empezando en la posición `off`. El primer byte leído se almacena en el elemento `b[off]`, el siguiente en `b[off+1]`, y así sucesivamente. Devuelve el número de bytes realmente leídos. Intenta leer al menos un byte. Si no hay ningún byte disponible porque la secuencia está al final del archivo, se devuelve el valor -1; de lo contrario, se lee y almacena al menos un byte en b. Puede lanzar una excepción del tipo **IOException**.
`void close()`	Cierra el InputStream. Puede lanzar una excepción del tipo **IOException**.

Métodos para escritura: OutputStream	
Método	**Descripción**
`void write(int b)`	Escribe un byte (los últimos 8 bits del int pasado como parámetro) en el OutputStream. Puede lanzar una excepción del tipo **IOException**.
`void write(byte[] b)`	Escribe todos los bytes del array `b` en el OutputStream. Puede lanzar una excepción del tipo **IOException**.
`void write(byte[] b, int offset, int len)`	Escribe un total de `len` bytes desde la posición `offset` del array `b`. Por ejemplo, si el array es de longitud 10 y se especifica un `offset` de 2 y un `len` de 5, se escribirán los bytes de `b[2]` a `b[6]`. Si `len` es mayor que la cantidad de bytes restantes en el array (desde `offset`), se pueden producir errores o comportamientos inesperados. Puede lanzar una excepción del tipo **IOException**.
`void flush()`	Vacía el flujo de salida actual del OutputStream y fuerza la escritura de todos los bytes que haya en él. Puede lanzar una excepción del tipo **IOException**.
`void close()`	Cierra el OutputStream. Puede lanzar una excepción del tipo **IOException**.

Ejemplo 07: Lectura/escritura en ficheros binarios con FileInputStream y FileOutputStream




```java
/*****************************************************************************
 * Escribe los números del 1 al 100 byte a byte en un fichero, y después
 * lee el fichero byte a byte y lo muestra por consola
 *****************************************************************************/
public class UT1_24_25_EjemploEscribeLeeFichBytes {
    public static void main(String[] args) {

        File ruta = new File("fichBytes.dat");

        int i;

        //try-with-resources: se crean el stream de salida FileOutputStream y
        //el stream de entrada FileInputStream, que se cerrarán automáticamente
        try (FileOutputStream flujoSalida = new FileOutputStream(ruta);
                FileInputStream flujoEntrada = new FileInputStream(ruta)) {

            // Escribimos los números del 1 al 100 byte a byte en el fichero
            for (i = 1; i <= 100; i++) {
                flujoSalida.write(i);
            }

            // Leemos el fichero byte a byte
            while ((i = flujoEntrada.read()) != -1) {  // -1, Fin de fichero
                System.out.println(i);
            }

        } catch (IOException ioe) {
            System.out.println("ERROR en clase: " +
                    UT1_24_25_EjemploEscribeLeeFichBytes.class.getSimpleName()+": "+ioe);
        }//Fin try-catch//Fin try-catch
    }//Fin main
}//Fin clase
```

➢ Ejercicios 13, 14, 15 y 16 de la hoja de ejercicios UT1 Parte 2

Ejemplo 08: Leer/escribir tipos primitivos y Strings en ficheros binarios con DataInputStream y DataOutputStream

- Las clases `DataInputStream` y `DataOutputStream` permiten leer y escribir tipos de datos primitivos de manera más eficiente en archivos binarios. La información se lee/escribe en forma binaria.

DataInputStream

- **Constructores**: reciben como parámetro cualquier subclase de InputStream.

  Ejemplo:

```
FileInputStream fileInput = new FileInputStream("datos.bin");
DataInputStream dataInput = new DataInputStream(fileInput);
```

- **Métodos** para leer tipos primitivos como `readchar(), readDouble(), readInt(),` readUTF(), etc.:

  Formato genérico: `public XXX readXXX()`

  Donde XXX representa un tipo de datos básico. Hay un método `read` para cada tipo de dato básico: readChar, readInt, readDouble, readUTF, etc. Puede lanzar una excepción del tipo **IOException.**

- Para leer todo el contenido de un archivo binario con los métodos de DataInputStream hay que tener en cuenta el tipo de datos que se está leyendo, y controlar cuándo se llega al final del archivo (EOF, End of File). Normalmente se utiliza un bucle while (true) para intentar leer datos hasta que se alcance el final del archivo, y se captura la excepción EOFException :

```
try(DataInputStream dis=new DataInputStream(new FileInputStream(nombreFichero)))
{
        while (true) {
                try {
                    // Leer datos del archivo, según su tipo
                    ...
                } catch (EOFException e) {
                    // Se alcanza el final del archivo
                   ...
                    break; // Salir del bucle cuando se detecte EOF
                }
        }
    } catch (IOException e) {
            e.printStackTrace();
}
```

Ejemplo: leer datos en forma de enteros desde un archivo binario hasta que se llega al final del archivo (EOF)

```
public class LeerFicheroBinario {

    public static void main(String[] args) {
        String nombreFichero = "datos.bin";  // Nombre del archivo binario que vamos a leer

        try (DataInputStream dis = new DataInputStream(new FileInputStream(nombreFichero))) {
            while (true) {
                try {
                    // Lee un entero desde el archivo
                    int dato = dis.readInt();
                    System.out.println("Dato leído: " + dato);
                } catch (EOFException e) {
                    // Se alcanza el final del fichero
                    System.out.println("Fin del archivo.");
                    break;  // Salimos del bucle
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

DataOutputStream

- **Constructores**: reciben como parámetro cualquier subclase de OutputStream.

  Ejemplo:

  ```
  FileOutputStream fileOutput = new FileOutputStream("datos.bin");
  DataOutputStream dataOutput = new DataOutputStream(fileOutput);
  ```

- **Métodos** para escribir tipos primitivos como `writechar(), writeDouble(), writeInt(),` writeUTF(), etc.:

  Formato genérico: `public final void writeXXX(argumento)`

  Donde XXX representa un tipo de datos básico. Hay un método `write` para cada tipo de dato básico: writeChar, writeInt, writeDouble, writeUTF, etc. Puede lanzar una excepción del tipo **IOException.**

```java
/* ************************************************************************
 * Guarda en un archivo de manera binaria los siguientes datos:
 * Un entero (25), un double (7.5), un booleano (true) y una cadena de texto
 * ("Hola Mundo!"). A continuación lee todos los datos desde ese archivo y los
 *  muestra en la pantalla de la siguiente manera:
 *      Entero: 25
 *      Double: 7.5
 *      Boolean: true
 *      Cadena: Hola Mundo!
 ************************************************************************/
public class UT1_24_25_EjemploDataInputStreamDataOutputStream {

    public static void main(String[] args) {
        // Nombre del archivo
        String fileName = "datos.bin";

        // Escribe datos en el archivo binario
        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(fileName))) {
            dos.writeInt(25);     // Escribe un entero
            dos.writeDouble(7.5); // Escribe un double
            dos.writeBoolean(true); // Escribe un boolean
            dos.writeUTF("Hola Mundo!"); // Escribe una cadena de texto en UTF-8
        } catch (IOException ioe) {
            System.out.println("ERROR en clase " +
                    UT1_24_25_EjemploDataInputStreamDataOutputStream.class.getSimpleName()+": "+ioe);
        }

        // Lee datos del archivo binario
        try (DataInputStream dis = new DataInputStream(new FileInputStream(fileName))) {
            int numero = dis.readInt();         // Leer un entero
            double decimal = dis.readDouble();  // Leer un double
            boolean valor = dis.readBoolean();  // Leer un boolean
            String mensaje = dis.readUTF();     // Leer una cadena en UTF-8

            // Muestra los datos leídos
            System.out.println("Entero: " + numero);
            System.out.println("Double: " + decimal);
            System.out.println("Boolean: " + valor);
            System.out.println("Cadena: " + mensaje);
        } catch (IOException ioe) {
            System.out.println("ERROR en clase " +
                    UT1_24_25_EjemploDataInputStreamDataOutputStream.class.getSimpleName()+": "+ioe);
        }
    }
}
```

Ejemplo de ejecución:

```
run:
Entero: 25
Double: 7.5
Boolean: true
Cadena: Hola Mundo!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo 09: Leer/escribir objetos en ficheros binarios con ObjectInputStream y ObjectOutputStream

- Las clases **`ObjectInputStream/ObjectOutputStream`** se utilizan para leer y escribir objetos en un archivo, mediante la serialización/deserialización.
- La **serialización** es el proceso de convertir un objeto en una secuencia de bytes para que pueda ser almacenado en un archivo o transmitido a través de una red. Permite que el estado del objeto se guarde de manera que pueda ser recreado más tarde mediante el proceso inverso (**deserialización)**.
- La clase del objeto que se quiere serializar debe implementar la interfaz Serializable.

```
class NombreClase implements Serializable { … }
```

Métodos principales

- **ObjectInputStream**:

  `Object readObject()`: Lee un objeto del flujo.

- **ObjectOutputStream:**

  `void writeObject(Object obj)`: Escribe un objeto serializable en el flujo.

Estructura de un archivo serializado en Java

Los archivos que contienen objetos serializados en Java siguen una estructura específica:

1. **Magic Number:** es un número especial (una secuencia específica de bytes) que se incluye automáticamente **al inicio de un archivo o stream** `para identificar que el archivo contiene datos serializados`. Cuando un archivo se abre para deserialización (lectura), lo primero que hace ObjectInputStream es comprobar este magic number para asegurarse de que el archivo está en el formato correcto. Si este número no coincide con lo esperado, lanzará una excepción (StreamCorruptedException).
2. **Versión del formato de serialización:** indicar la versión del protocolo de serialización utilizado.
3. **Datos de los objetos:** metadatos de la clase, atributos del objeto, y valores de los atributos. Cada vez que Java serializa un objeto, también escribe información sobre la clase del objeto (metadatos), lo que incluye:
   - El nombre completo de la clase (incluido el paquete).
   - La estructura de la clase (como los nombres y tipos de los atributos).
   - Superclase y Jerarquía de Herencia: si la clase que se serializa hereda de otra clase que también es Serializable, los metadatos de la superclase también se almacenan en el archivo. Esto asegura que, al deserializar el objeto, se pueda reconstruir toda la jerarquía de la clase.
   - El valor de `serialVersionUID` (SUID) de la clase.

   Esta información de la clase (los metadatos) sólo se escribe la primera vez que un objeto de esa clase es serializado en un flujo de datos. Si varios objetos de la misma clase se serializan en el mismo stream, no se vuelven a escribir los metadatos de la clase, sino que Java guarda una referencia a la información de clase previamente escrita.

   El `serialVersionUID` es un identificador único de versión que se asocia a las clases que implementan la interfaz Serializable. Este SUID permite que, durante la deserialización, el sistema pueda verificar que la versión de la clase utilizada para escribir el objeto es compatible con la clase que se está utilizando para leerlo. Si el `serialVersionUID` de la clase que se utiliza para deserializar no coincide con el `serialVersionUID` de la clase original que se serializó, se lanzará una **`InvalidClassException`**.

- Definición manual: consiste en definir explícitamente este identificador dentro de la clase para mantener el control sobre la compatibilidad entre versiones de la clase. Es habitual usar el valor 1L para identificar la primera versión oficial de la clase que implementa la serialización:

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

  Si se hacen pequeños cambios en la clase (como añadir métodos que no afectan los atributos serializados), se debe mantener el mismo serialVersionUID, garantizando así que los objetos guardados previamente seguirán siendo válidos para deserialización. En cambio, si la clase cambia de manera que sea incompatible con las versiones anteriores (por ejemplo, si se eliminan o modifican campos importantes), se debe incrementar el valor del serialVersionUID (por ejemplo, a 2L, 3L, etc).

- Definición automática: si no se define el serialVersionUID, el compilador genera uno basado en los detalles de la clase (atributos, métodos, etc.). Este número puede cambiar si la clase sufre cualquier modificación, lo que podría causar problemas de compatibilidad en el futuro, ya que el objeto serializado podría no coincidir con la versión de la clase en tiempo de deserialización.

```java
/* *********************************************************************************************
 * Escribe un objeto de la clase Persona en un archivo binario, luego lo lee del
 * archivo y lo imprime en la consola.
 * La clase Persona implementa la interfaz Serializable para poder convertir sus
 * objetos en una secuencia de bytes, y así poder ser almacenados o transmitidos
 *
 * serialVersionUID: Es una constante que ayuda a verificar la compatibilidad
 * entre la versión del objeto serializado y la versión de la clase. Si la clase
 * cambia y no coincide el serialVersionUID, se puede lanzar una excepción InvalidClassException
 ********************************************************************************************* */
class Persona implements Serializable {

    private static final long serialVersionUID = 1L;
    String nombre;
    int edad;

    public Persona (String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}

public class UT1_24_25_EjemploObjectStreams {
    public static void main(String[] args) {
        // Escritura de objeto
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("persona.bin"))) {
            Persona p = new Persona("Carlos", 30);
            oos.writeObject(p);  // Escribimos el objeto en el archivo
        } catch (IOException e) {
           System.out.println("ERROR en clase: " +
                    UT1_24_25_EjemploObjectStreams.class.getSimpleName()+": "+e);
        }

        // Lectura de objeto
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("persona.bin"))) {
            Persona p = (Persona) ois.readObject();  // Leemos el objeto desde el archivo
            // Mostramos los atributos directamente sin usar toString()
            System.out.println("Nombre: " + p.nombre);
            System.out.println("Edad: " + p.edad + " años");
        } catch (IOException | ClassNotFoundException e) {
           System.out.println("ERROR en clase: " +
                    UT1_24_25_EjemploObjectStreams.class.getSimpleName()+": "+e);
        }

    }
}
```

Ejemplo de ejecución:

```
run:
Carlos, 30 años
Nombre: Carlos
Edad: 30 años
BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Operaciones básicas con ficheros de acceso aleatorio: clase RandomAccessFile

El acceso aleatorio a ficheros en Java permite leer y escribir en cualquier posición dentro de un archivo, sin necesidad de seguir un orden secuencial desde el inicio, permitiendo modificar solo una parte del archivo o leer datos específicos sin tener que procesar todo el contenido. Esta clase es especialmente útil para trabajar con estructuras de datos que necesitan acceder a posiciones específicas dentro de un archivo, como bases de datos indexadas o archivos con registros de tamaño fijo.

La clase **`RandomAccessFile`** en `java.io` proporciona este tipo de acceso aleatorio o acceso directo, permitiendo moverse libremente por el archivo, tanto para leer como para escribir. Esta clase combina las funcionalidades de las clases `InputStream` y `OutputStream`, ofreciendo métodos para leer y escribir datos primitivos (como enteros, cadenas o caracteres), pero con la ventaja de que es posible posicionarse en cualquier parte del archivo con un **puntero de archivo**. Este puntero indica la posición actual en el archivo, y se desplaza por el archivo al realizar operaciones de lectura/escritura. También es posible desplazar el puntero libremente por el fichero usando los métodos `seek` o `skipBytes`.

`RandomAccessFile` permite dos modos de acceso al fichero, lectura y lectura/escritura.

Ventajas de RandomAccessFile:

- Permite leer y escribir datos en cualquier posición dentro de un archivo.
- Útil para modificar archivos grandes sin necesidad de cargarlos completamente en memoria.
- Soporte para la lectura y escritura de tipos de datos primitivos directamente.

Desventajas:

- Al ser una clase más básica que otras como BufferedReader o BufferedWriter, RandomAccessFile no ofrece buffering interno, lo que puede hacer que las operaciones de lectura/escritura sean más lentas.
- Dificultad para manejar archivos grandes o complejos si no se planifica bien el uso del puntero de archivo.

Constructores

- `RandomAccessFile(String name, String mode)`
- `RandomAccessFile(File file, String mode)`

Donde `mode` especifica el modo de apertura del fichero:

- **"r"**: Abre el archivo en modo solo lectura.
- **"rw"**: Abre el archivo en modo lectura y escritura.

Métodos principales

Métodos de posicionamiento en el fichero	
Método	**Descripción**
`long getFilePointer()`	Devuelve la posición actual del puntero dentro del fichero. Indica la posición (en bytes) donde se va a leer o escribir. Puede lanzar una excepción del tipo **IOException.**
`long length()`	Devuelve la longitud total del fichero en bytes. Puede lanzar una excepción del tipo **IOException.**
`public void setLength(long newLength)`	Establece la longitud del archivo al valor `newLength`. Puede lanzar una excepción del tipo **IOException.** • Si la longitud actual del archivo devuelta por el método `length()` es mayor que el valor `newLength`, el archivo se truncará. En este caso, si la posición del puntero del archivo era mayor que `newLength`, después de ejecutarse `setLength` la posición del puntero será igual a `newLength`. • Si la longitud actual del archivo devuelta por el método `length()` es menor que el valor `newLength`, entonces el archivo se ampliará hasta la nueva longitud. En este caso, el contenido de la parte ampliada del archivo no está definido.
`void seek(long pos)`	Coloca el puntero del fichero en una posición `pos` determinada. La posición se da como un desplazamiento **en bytes** desde el comienzo del fichero: • La posición `0` indica el inicio del fichero • La posición `length()` indica el final del fichero Puede lanzar una excepción del tipo **IOException** si `pos` es menor que `0`, o si ocurre un error de entrada/salida.
`public int skipBytes(int n)`	Intenta saltar n bytes del archivo, descartando los bytes omitidos. Es posible que se omitan un número de bytes menor que n (incluso 0), por ejemplo, si se alcanza el final del archivo antes de que se hayan omitido n bytes. Este método nunca arroja una EOFException. Se devuelve el número real de bytes omitidos. Si n es negativo, no se omite ningún byte. Puede lanzar una excepción del tipo **IOException.**

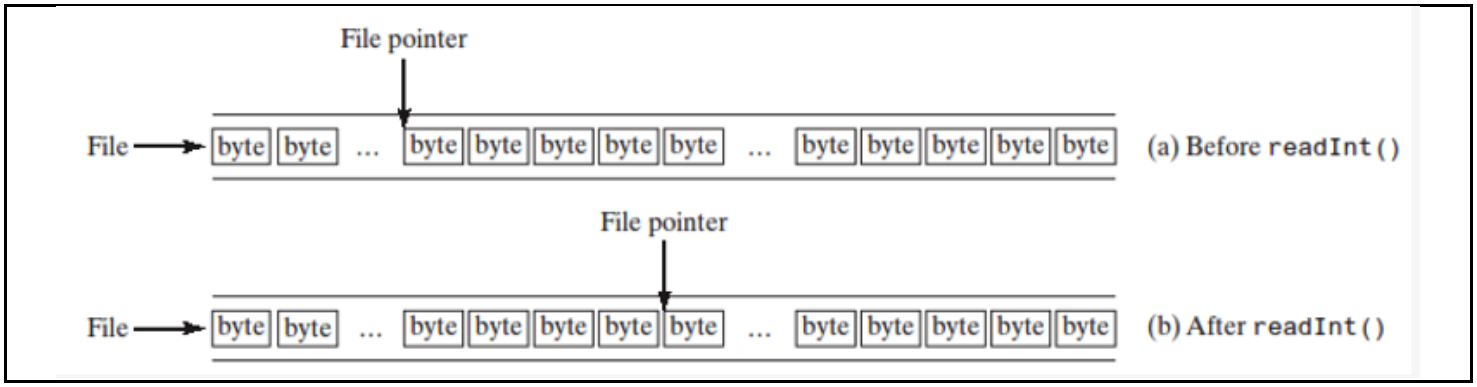
Métodos de escritura	
Método	**Descripción**
`public void write(int b)`	Escribe en el fichero el byte que se le pasa como parámetro, comenzando en la posición actual del puntero. Se debe utilizar este método para escribir caracteres en un fichero de texto. Puede lanzar una excepción del tipo **IOException.**
`void write(byte[] b)`	Escribe en el fichero todos los bytes del array de bytes que se le pasa como parámetro, comenzando en la posición actual del puntero. Puede lanzar una excepción del tipo **IOException**.
`void write(byte[] b, int offset, int len)`	Escribe en el fichero un total de `len` bytes desde la posición `offset` del array `b` que se le pasa como parámetro, comenzando en la posición actual del puntero. Puede lanzar una excepción del tipo **IOException**.
`public final void writeXXX(argumento)`	Donde XXX representa un tipo de datos básico. Existe un método write para cada tipo de datos básico: writeChar, writeInt, writeDouble, writeUTF, etc. Puede lanzar una excepción del tipo **IOException.**
`public final void writeBytes(String s)`	Escribe en el fichero el String que se le pasa como parámetro como una secuencia de bytes (cada carácter se escribe con 1 byte, descartando los 8 bit superiores). Puede lanzar una excepción del tipo **IOException.**
`public final void writeChars(String s)`	Escribe en el fichero el String que se le pasa como parámetro como una secuencia de caracteres (cada carácter ocupa 2 bytes). Puede lanzar una excepción del tipo **IOException.**
`public final void writeUTF(String s)`	Escribe en el fichero el String que se le pasa como parámetro utilizando codificación UTF-8 modificada de forma independiente de la máquina. Primero se escriben dos bytes que indican el número de bytes que se van a escribir (equivale a llamar a writeShort(longitud), donde longitud representa el número de bytes necesarios para codificar la cadena), y a continuación se escribe la secuencia de caracteres del String. Puede lanzar una excepción del tipo **IOException.**

Métodos de lectura	
Método	**Descripción**
`public int read()`	Devuelve el byte leído en la posición marcada por el puntero. Devuelve -1 si alcanza el final del fichero. Se debe utilizar este método para leer caracteres de un fichero de texto. Puede lanzar una excepción del tipo **IOException.**
`int read(byte[] b)`	Se comporta igual que el método homónimo de la clase InputStream.
`int read(byte[] b, int off, int len)`	Se comporta igual que el método homónimo de la clase InputStream.
`public final String readLine()`	Devuelve la cadena de caracteres que se lee, desde la posición marcada por el puntero, hasta el siguiente salto de linea '\n' o retorno de carro '\r', sin incluir estos caracteres en la cadena devuelta. Puede lanzar una excepción del tipo **IOException.**
`public XXX readXXX()`	Donde XXX representa un tipo de datos básico. Hay un método read para cada tipo de datos básico: readChar, readInt, readDouble, readUTF, etc. Puede lanzar una excepción del tipo **IOException.**

Tamaños en bytes de diferentes tipos de datos

TIPO	DESCRIPCIÓN	DEFAULT	TAMAÑO	EJEMPLOS
boolean	true o false	false	1 bit	true, false
byte	entero complemento de dos	0	1 byte (8 bits)	100, -50
char	carácter unicode	\u0000	2 bytes (16 bits)	'a', '\u0041', '\101', '\\'
short	entero complemento de dos	0	2 bytes (16 bits)	10000,-20000
int	entero complemento de dos	0	4 bytes (32 bits)	100000,-2,-1,0,1,2,-200000
long	entero complemento de dos	0	8 bytes (64 bits)	-2L,-1L,0L,1L,2L
float	coma flotante IEEE 754	0.0	4 bytes (32 bits)	1.23e100f, -1.23e-100f, .3ef, 3.14f
double	coma flotante IEEE 754	0.0	8 bytes (64 bits)	1.2345e300d, -1.2345e-300f, 1e1d

Ejemplo gráfico: lectura de un entero (4 bytes) con readInt()



Ejemplo 10: Ejemplo básico de lectura/escritura con RandomAccessFile

```java
/**********************************************************************************************************
 * Ejemplo básico de la clase RandomAccessFile.
 * Abre un archivo de acceso aleatorio para lectura/escritura, escribe algunos datos, los lee y
 * los muestra. Luego fuerza un error por mal posicionamiento del puntero, que genera una EOFException
 **********************************************************************************************************/
public class UT1_24_25_EjemploBasicoRandomAccessFile {
    public static void main(String[] args) {

        try ( // Abrimos el archivo en modo lectura/escritura
                RandomAccessFile raf = new RandomAccessFile("archivo.bin", "rw")) {

            // Escribimos datos en el archivo
            raf.writeInt(100);  // Escribe un entero de 4 bytes
            raf.writeDouble(3.14);  // Escribe un double de 8 bytes

            // Movemos el puntero al principio del archivo
            raf.seek(0);

            // Leemos el entero y el double
            int numero = raf.readInt();
            double pi = raf.readDouble();

            // Imprimimos los valores leídos
            System.out.println("Número: " + numero);
            System.out.println("PI: " + pi);

            // Vamos a ver un ejemplo de error al reposicionar el puntero con un offset y
            //escribir en el archivo
            raf.seek(4); //posiciona el puntero despés del int
            raf.writeChar('x'); //Escribe un char después del int, sobreescribiendo los
                                //primeros 2 bytes del double

            //Leemos de nuevo el fichero desde el principio, intentamos leer un int, un
            //char y un double
            raf.seek(0);
            numero = raf.readInt();//ok
            char car = raf.readChar();//ok
            pi = raf.readDouble();  //Intenta leer 8 bytes después del char, pero sólo
                                    //quedan 4, por tanto se alcanza EOF

            // Imprimimos los valores leídos
            System.out.println("Número: " + numero);
            System.out.println("Letra: " + car);//ok
            System.out.println("PI: " + pi);
        }

        catch (IOException ioe) {
            System.out.println("ERROR: "+ioe);
        }
    }
}
```

Ejemplo de ejecución:

```
run:
Número: 100
PI: 3.14
ERROR: java.io.EOFException
BUILD SUCCESSFUL (total time: 0 seconds)
```