IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma

UNIDAD 1: Gestión de Ficheros

Índice

1. Manejo básico de ficheros en java **2**
1.1 Concepto de fichero **2**
1.2 Clasificación de los ficheros **2**
1.3 Tipos de operaciones en ficheros **3**
1.4 Paquetes java relacionados con la entrada/salida **4**
1.5 Paquete java.io **6**
1.5.1 Stream (Flujos de datos) 6
1.5.2 Tipos de flujos en java 7
1.5.3 Utilización de los flujos 7
1.5.4 Los nombres de las clases de java.io 8
1.5.5 Flujos de bytes 8
1.5.6 flujos de caracteres 9
2. E/S estándar **10**
3. Sistema de Ficheros y Directorios en Java **11**
3.1 La clase FILE **11**
3.1.1 Creación de un objeto File 11
3.1.2 Métodos de la clase File 12
3.2 Java NIO **14**
3.2.1 Paquetes y clases principales de java.nio.file 14
3.2.2 La Clase Path 16
3.2.3 La Clase de Utilidad Paths 17
3.2.4 La Clase de Utilidad Files 17
3.2.5 Tipos de recorrido de directorios: Stream<Path> y DirectoryStream<Path> 21
3.2.6 FileSystem 25
3.2.7 FileVisitor<T> y SimpleFileVisitor<T>: Recorrido recursivo 26
3.2.8 BasicFileAttributes 27
4. Manejo de archivos en Java: lectura y escritura de datos **28**
4.1 Apertura y cierre de ficheros en Java **28**
4.2 Clases FileOutputStream y FileInputStream **29**
4.3 Clases FileReader y FileWriter **31**
4.3.1 Codificación de caracteres y errores habituales 33
4.4 Las clases OutputStreamReader e InputStreamReader. **35**
4.5 Buffer para el flujo de caracteres: Clases BufferedReader y BufferedWriter **37**
5. Acceso a datos primitivos:Clases DataInputStream y DataOuputStream **38**
5.1 Clases BufferedInputStream y BufferedOutputStream **40**
6. Serialización **41**
6.1 Interfaz Serializable **42**
6.2 Excluir campos al serializar objetos **42**
6.3 Campos estáticos (static) y serialización **42**
6.4 Qué se guarda exactamente al serializar un objeto **43**
6.5 Flujos para la entrada y salida de objetos: ObjectInputStream e ObjectOutputStream **45**
6.5.1 Escritura de objetos en ficheros 46
6.5.2 Lectura de objetos en ficheros 47
6.5.3 Serialización de objetos compuestos 48
6.5.4 Serialización de objetos con herencia 50

1. Manejo básico de ficheros en java

Los programas usan variables para almacenar información: los datos de entrada, los resultados calculados y valores intermedios generados a lo largo del cálculo. Toda esta **información es efímera, cuando salíamos del programa, todo lo que habíamos generado se pierde**. A veces nos interesaría que la vida de los datos fuera más allá que la de los programas que los generaron. Es decir, que al salir de un programa, los datos generados quedaran guardados en algún lugar que permitiera su recuperación desde el mismo u otros programas. Por tanto, querríamos que dichos datos fueran **persistentes**.

Persistencia: Es la capacidad que tiene el programador para que sus **datos se conserven al finalizar la ejecución de un proceso**, de forma que se puedan recuperar y reutilizar en otros procesos.

Cuando se desea **guardar información más allá del tiempo de ejecución** de un programa, podemos optar por las siguientes posibilidades:

- Organizar esa información en uno o varios **ficheros** almacenados en algún soporte de almacenamiento persistente.
- Almacenar los datos en una **base de datos**.

En este capítulo veremos el uso básico de archivos en Java para conseguir persistencia de datos. Para ello, presentaremos conceptos básicos sobre archivos y algunas de las clases de la biblioteca estándar de Java para su creación y manipulación.

1.1 Concepto de fichero

Un archivo o fichero:

es **una colección de datos homogéneos almacenados en un soporte físico del computador.**

- **Datos homogéneos**: Almacena colecciones de datos del mismo tipo (igual que arrays/vectores)

  Cada elemento almacenado en un fichero se denomina **registro**, que se compone de campos.

- Puede ser **almacenado en diversos soportes** (disco duro, disquete, pendrive,..)

Desde el **punto de vista de más bajo nivel**, podemos definir un archivo (o fichero) como:

Un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica. Es decir, un conjunto de 0s y 1s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD, entre otros.

1.2 Clasificación de los ficheros

Podemos utilizar varios criterios para distinguir diversas subcategorías de archivos.

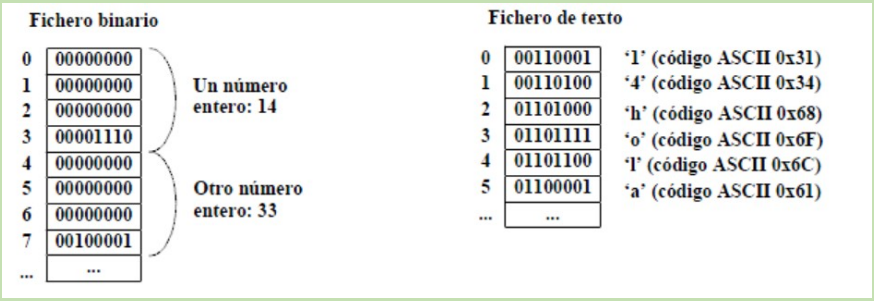
Estos tipos de archivos se **diferenciarán desde el punto de vista de la programación**, en que cada **uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación**.

- **Según su contenido**:

  Caracteres (texto) o bytes (binarios)

  Sabemos que es diferente manipular números que Strings, aunque en el fondo ambos acaben siendo bits en la memoria del ordenador.

Por eso, cuando manipulamos archivos, distinguiremos dos clases de archivos dependiendo del tipo de datos que contienen:



- **fichero de carácteres (Texto)** :

Formado exclusivamente **por caracteres** y que, por tanto, puede crearse y visualizarse usando un editor. Las operaciones de lectura y escritura trabajarán con caracteres.

Por ejemplo, los ficheros con código java son ficheros de texto.

- **fichero bytes (Binario)** ya no está formado por caracteres sino que **los** bytes que contiene pueden representar otras cosas como números, imágenes, sonido, etc.

NOTA: Para "entender" los contenidos de un fichero es necesario conocer de antemano el tipo de datos que contiene.

- **Según el tipo de acceso a los datos**

Existen dos modos básicos de acceso a la información contenida en un archivo:

Secuencial o Acceso aleatorio

- **Acceso Secuencial:**

Datos son leídos secuencialmente, desde el principio hasta el final.

La información del archivo es una secuencia de bytes (o caracteres) de manera que para acceder al byte (o carácter) N se ha de haber accedido con anterioridad a los N-1 anteriores.

- **Acceso aleatorio ("Random" o directo):**

Permiten acceder a los datos en forma no secuencial, desordenada.

Se puede **acceder directamente a la información del byte N**.

1.3 Tipos de operaciones en ficheros

Las **operaciones más comunes en ficheros** son:

- Operación de **Creación**
- Operación de **Apertura**. Varios modos:
  - Sólo lectura
  - Sólo escritura
  - Lectura y Escritura
- Operaciones de **lectura / escritura**
- Operaciones de **inserción / borrado**
- Operaciones de **renombrado / eliminación**
- Operación de **desplazamiento** dentro de un fichero
- Operación de **ordenación**
- Operación de **cierre**

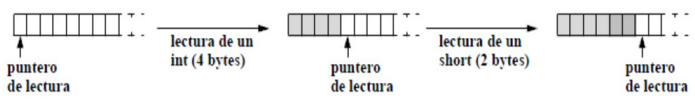
Punteros de lectura y escritura:

Las **operaciones sobre fichero se van a realizar en el byte** que señalen **los punteros de lectura y escritura**:

• Indican el **próximo byte a leer o a escribir.**

• **Gestionados automáticamente** por el sistema operativo

Ejemplo:



• Cuando se **abren, se comienzan apuntando al primer byte** del fichero

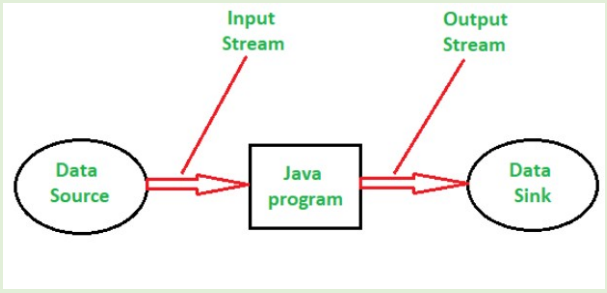
• Van **avanzando por el fichero según se van leyendo** sus contenidos o **escribiendo** datos.

1.4 Paquetes java relacionados con la entrada/salida

Paquete io.java

- **Paquete estándar** que tiene todas las clases necesarias que podemos usar para las **operaciones de E/S de Java**.
- Está **orientado a stream** (flujos, secuencias). Utiliza flujos para transferir los datos entre la fuente o receptor de datos y el programa Java. Los datos fluyen de un punto a otro (Fuente → Programa → Destino).
- Es una **transferencia de datos unidireccional**. Un flujo de entrada (InputStream) o un flujo de salida (OutputStream).

La siguiente imagen ilustra un paquete orientado a la transmisión:



- **E/S de bloqueo (o síncrono**).

  Esto significa que si un proceso invoca una operación de lectura() o escritura(), **ese proceso se bloquea hasta que la operación finaliza**.

- **Clases principales:**

Tipo de flujo	Entrada (Input)	Salida (Output)
Bytes	`FileInputStream`, `BufferedInputStream`, `DataInputStream`	`FileOutputStream`, `BufferedOutputStream`, `DataOutputStream`
Caracteres	`FileReader`, `BufferedReader`, `InputStreamReader`	`FileWriter`, `BufferedWriter`, `PrintWriter`
Objetos	`ObjectInputStream`	`ObjectOutputStream`
Acceso aleatorio	`RandomAccessFile` (lectura y escritura)	`RandomAccessFile` (lectura y escritura)

Paquete nio.java

- **Java New Input/Output (NIO)** es una estructura de **manejo de archivos de alto rendimiento y una API de red.**
- Complementa a java.io: No reemplaza las clases de entrada/salida (IO) basadas en secuencias en el paquete java.io. Funciona como una **alternativa a java.io con algunas características y funcionalidades adicionales** para trabajar con Entrada/Salida (IO) de manera diferente.
- El NIO de Java se introdujo a partir de la versión JDK 4.
- Paquete **orientado a búffer**.

  Los datos **se leen primero en un búfer (buffer) y luego se procesan desde ahí utilizando un canal.**

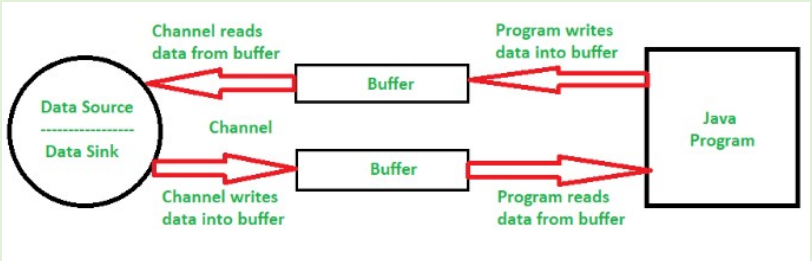
  Permite **manipular los datos de manera más flexible y eficiente** que con flujos secuenciales (Streams) de java.io.
- **E/S sin bloqueo (o asíncrono**).

  Un proceso puede pedir a un canal que lea datos mientras realiza otras tareas.

  Una vez que los datos llegan al búfer, el proceso continúa con la operación pendiente.

  Por ejemplo, un proceso pide a un canal que lea datos en un búfer. Mientras el canal está leyendo datos en el búfer simultáneamente, el proceso puede realizar algún otro trabajo. Una vez que los datos se leen en el búfer, el proceso puede continuar gestionando el trabajo que había dejado durante la operación de lectura.
- NIO es una **transferencia de datos bidireccional.**

  Los **canales permiten leer y escribir datos, no solo en un sentido**.



Se utilizan:

- **Canales (Channel)**:

  Un canal es un **medio para la transmisión de datos eficiente entre la fuente o receptor y el búfer**.

Actúa como una puerta de enlace para una conexión abierta con la fuente / receptor de datos

- **Selector**: selecciona el canal entre los múltiples canales IO usando un proceso único.

Tipo de operación	Canales principales	Selector
Lectura de bytes	`FileChannel`, `SocketChannel`	Detecta canales listos para leer
Escritura de bytes	`FileChannel`, `SocketChannel`	Detecta canales listos para escribir
Lectura/escritura de caracteres	`SocketChannel` + `CharsetDecoder/Encoder`	Selecciona canales listos para procesar caracteres
Acceso aleatorio	`FileChannel` (position(), map())	Se puede gestionar múltiples archivos combinando con `AsynchronousFileChannel`

Notas clave resumidas:

- Todos los canales usan **buffers** (`ByteBuffer` o `CharBuffer`).
- Los selectores permiten manejar **múltiples canales con un solo hilo.**

1.5 Paquete java.io

El paquete **java.io** contiene todas **las clases relacionadas con las funciones de entrada (input) y salida (output)**.

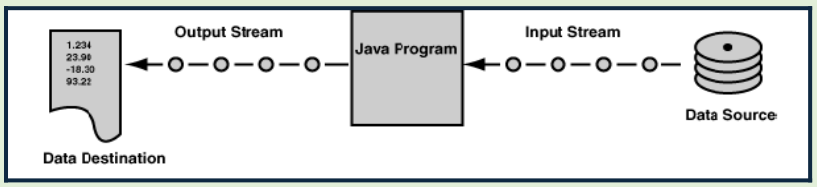
1.5.1 Stream (Flujos de datos)

Java se basa en ***las secuencias de datos*** para dar facilidades de entrada y salida.

- *Una secuencia es una **corriente de datos** en serie entre **un emisor** y un **receptor** de datos en cada extremo.*

Los programas en Java realizan la **E/S** a través de **streams (Flujos)**, es decir, cualquier programa realizado en Java que necesite llevar a cabo una operación de E/S lo hará a través de un stream.

Un stream, cuya traducción literal es "*flujo*", es una abstracción de todo aquello que produzca o consuma información



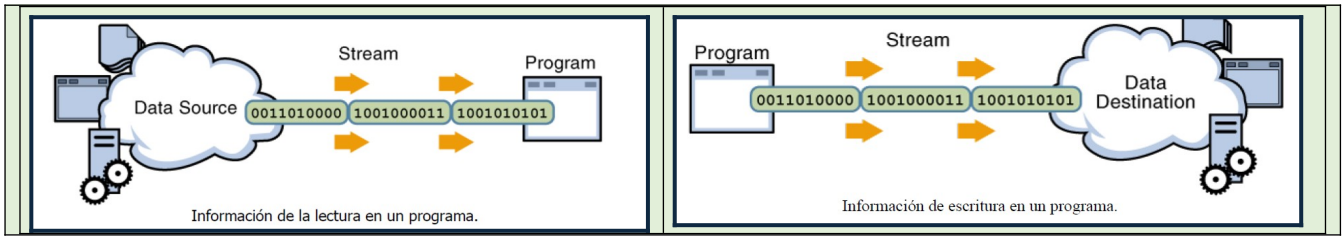
- **Un flujo no está relacionado con un dispositivo físico**, comportándose todos los flujos de la misma manera, aunque traten con dispositivos distintos.

  Debido a esta característica **se pueden aplicar las mismas clases a cualquier tipo de dispositivo.**

- **La vinculación del stream al dispositivo físico lo lleva a cabo el sistema de entrada y salida de Java**.

  Las clases y métodos de I/O que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando, luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de ficheros o un socket de red liberando a nuestro código de tener que saber con quién está interactuando.

Se utiliza **input stream** (un flujo de entrada) para **obtener los datos** leídos de una fuente:	Se utiliza un **output stream** (flujo de salida) para **escribir los datos** a un destino:



La fuente de datos (data source) y el destino de los datos (data destination) de los diagramas anteriores puede ser cualquier cosa que genera o consume datos. Esto incluye obviamente archivos de disco, pero una fuente o un destino puede ser otro programa, un dispositivo (impresora, periférico, etc.), un socket de red (es un método para la comunicación entre un programa del cliente y un programa del servidor en una red) o un array.

1.5.2 Tipos de flujos en java

Podemos distinguir básicamente dos tipos de flujos:

- **Los flujos de bytes: Los datos fluyen en serie, byte a byte.**
- Los **flujos de caracteres**: Transmiten **caracteres Java.**

Flujos de bytes	Flujos de caracteres
▪ Proporciona un medio adecuado para el manejo de **entradas y salidas de bytes** ▪ Su uso está orientado a la **lectura y escritura de datos binarios.** ▪ **Clases abstractas** para el manejo de flujos de bytes: **InputStream y OutputStream**. *Cada una de estas clases abstractas tiene varias subclases concretas que controlan las diferencias entre los distintos dispositivos de I/O que se pueden utilizar.* ▪ Definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan: • **read ()** para **la lectura** de bytes de datos. • **write ()** para la **escritura** de bytes de datos.	▪ Proporciona un medio conveniente para el manejo de **entradas y salidas de caracteres**. ▪ Su uso está orientado a manejar **caracteres**, *Al nivel más bajo todas las operaciones de I/O son orientadas a byte.* ▪ Usan **codificación Unicode** (por tanto, se pueden internacionalizar). ▪ **Clases abstractas** para el manejo de caracteres: **Reader y Writer**. ▪ Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados: • **read ()** para la **lectura** de caracteres. • **write ()** para la **escritura** de caracteres.

1.5.3 Utilización de los flujos

La **utilización de los flujos** es un nivel de abstracción que **hace que un programa no tenga que saber nada del dispositivo**, lo que se traduce en una facilidad más a la hora de escribir programas, ya que los algoritmos para leer y escribir datos serán siempre más o menos los mismos.

En general, las operaciones serán:

Lectura	Escritura
Abrir un flujo a una fuente de datos Se crea un objeto stream: Teclado, Fichero, Socket	**Abrir un flujo** a una fuente de datos Se crea un objeto stream: Pantalla, Fichero, Socket

▪ **Mientras existan datos disponibles** • **Leer datos** ▪ **Cerrar el flujo** (método close)	▪ **Mientras existan datos disponibles** • **Escribir datos** ▪ **Cerrar el flujo** (método close)
• Un **fallo en cualquier punto** produce la **excepción IOException.** • Para los **flujos estándar: se encarga el sistema de abrirlos y cerrarlos** (system.in system.ouput).	

 Nota: Los sockets son un sistema de comunicación entre procesos de diferentes máquinas de una red. Mas exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información.

1.5.4 Los nombres de las clases de java.io

Las clases de **java.io** siguen una nomenclatura sistemática que permite deducir su función a partir del as palabras que componen el nombre, tal como se describe en la siguiente tabla:

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de **bytes**
Reader, Writer	Lectura/Escritura de **caracteres**
File	**Archivos y directorios**
Buffered	**Buffer**
Filter	**Filtro sobre el stream**
Data	**Intercambio** de datos en formato propio de Java (**datos primitivos**)
Object	Persistencia de **objetos**
RandomAccesFile	**Acceso directo o aleatorio**

 *Nota: **Un buffer es un espacio de memoria intermedia**. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.*

1.5.5 Flujos de bytes

- Los programas **utilizan secuencias de bytes** para realizar la **entrada y salida de bytes (8-bits)**.
- Todas las **clases de flujos de bytes descienden** de las clases abstractas **InputStream y OutputStream**.
- Al **tratarse de clases abstractas**, **no vamos a poder crear objetos de estas clases** porque su funcionalidad está "incompleta" (tienen métodos que no están definidos, implementándolos las subclases), y así nos permiten representar un flujo de datos de entrada o salida binario cualquiera.
- **Cada una de las clases hijas de InputStream y OutputStream proporciona** una **funcionalidad más específica a la clase padre**, por lo tanto, **tendremos que crear objetos de alguna de sus subclases.**

Las clases **InputStream y OutputStream** son clases abstractas de las que se derivan las clases que permiten **crear flujos de bytes de 8 bits** de lectura e escritura.

Métodos de InputStream	
`int`	`available()` Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
`void`	`close()` **Cierra la corriente de entrada y libera los recursos del sistema que esté usando.**
`abstract int`	`read()` **Lee el siguiente byte de la corriente de entrada.**
`int`	`read(byte[] b)` **Lee bytes de la corriente de entrada y los deposita en el vector `b`.**

int	read(byte[] b, int off, int len) Lee hasta len bytes de la corriente de entrada y los deposita en b a partir de off.
void	reset() Coloca la corriente de entrada en la posición que tenía la última vez que se invocó mark.
long	skip(long n) Salta y descarta los siguientes n bytes de la corriente de entrada.

Métodos de OutputStream	
void	close() Cierra la corriente de salida y libera los recursos del sistema que está utilizando.
void	flush() Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
void	write(byte[] b) Escribe todos los b.length bytes del vector b en la corriente de salida.
void	write(byte[] b, int off, int len) Escribe len bytes del vector b comenzando en la posición off.
abstract void	write(int b) Escribe un byte especificado por el int b.

Los **métodos más importantes** son los que sirven para **leer y escribir: read y write.**

- Los tres **métodos read** sirven para leer bytes:

```
public abstract int read() throws IOException;
public int read(byte[] b) throws IOException;
public int read(byte[] b,int pos, int n) throws IOException;
```

El primero lee un solo byte y devuelve su valor como un entero entre 0 y 255. El segundo lee tantos bytes como pueda hasta llenar el vector (array) b que se le pasa como parámetro o hasta que se acabe la corriente. El último lee los n bytes y los pone en el vector b, a partir de la posición pos.

*¿Qué ocurre si la **corriente de entrada se termina**? En ese caso el resultado de los métodos **read es -1.***

- Los tres **métodos de escritura write** son análogos a los de lectura.

```
public abstract void write(int b) throws IOException;
public void write(byte[] b) throws IOException;
public void write(byte[] b,int pos, int n) throws IOException;
```

El primero escribe un byte aunque se le pase como parámetro un entero. El método convierte el entero a byte y luego lo escribe. Para no perder información al método write(int n) sólo se le deben pasar valores n entre 0 y 255. El segundo método write escribe todos los bytes del vector b que se le pasa como parámetro y el tercero escribe n bytes del vector b comenzando desde la posición pos.

1.5.6 flujos de caracteres

- Existen dos variaciones para las Clases InputStream y OutputStream que son: **Writer y Reader**.
- La principal diferencia entre estas clases es que las clases InputStream y OutputStream solamente soportan "Streams" de 8-bits byte, mientras las **Clases Writer y Reader soporta "Streams" de 16-bits.**
- La importancia de 16-bits radica en Java que utiliza **Unicode,**

  Al utilizarse 8-bits no es posible emplear muchos caracteres disponibles en Unicode, además debido a que las Clases Writer y Reader son una adición más reciente al JDK, estas poseen mayor velocidad de ejecución a diferencia de sus contrapartes InputStream y OutputStream.

Las clases **Reader y Writer** son clases abstractas de las que se derivan las clases que permiten **crear flujos de caracteres de 16 bits (Unicode)** de lectura e escritura.

Metodos de Reader	
abstract void	close() Cierra la corriente de entrada y libera los recursos del sistema que esté usando.
int	read() Lee un character.

int	**read(char[] cbuf)** **Lee caracteres de la corriente de entrada y los deposita en el vector de caracteres cbuf**
abstract int	**read(char[] cbuf, int off, int len)** **Lee len caracteres de la corriente de entrada y los deposita en el vector de caracteres cbuf a paritr de off**
boolean	**ready()** Indica si el flujo de entrada está listo para ser leído
void	**reset()** Restablece el flujo de entrada.
long	**skip(long n)** Salta n caracteres.
Métodos de Writer	
Writer	**append(char c)** Añade el caracter especificado en el fujo de entrada
abstract void	**close()** **cierra el flujo de salida**
abstract void	**flush()** **Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.**
void	**write(char[] cbuf)** **Escribe un array de caracteres**
abstract void	**write(char[] cbuf, int off, int len)** **Escrobe len caractees a partir del desplazamineto marcado por off.**
void	**write(int c)** **Escribe un caracerer**
void	**write(String str)** **Escribe una cadena**
void	**write(String str, int off, int len)** **Escribe una porción de la cadena empezando en off y de longitud len.**

2. E/S estándar

Todos los programas Java importan el paquete **Java.lang** *automáticamente*. Este paquete define una clase llamada **System**:

java.lang

Class System

java.lang.Object
  java.lang.System

```
public final class System
extends Object
```

Es una clase final y todos sus contenidos son privados.

- **No se puede instanciar un objeto de esa clase**.
- La clase System **siempre está ahí disponible para que se pueda invocar cualquiera de sus métodos.**
- Todos los miembros del paquete **java.lang** se pueden usar directamente, **no hay que importarlos**.

- Contiene tres variables **con flujos predefinidos** llamadas **in, out** y **err**, que se pueden utilizar sin tener una referencia a un objeto System específico.

En Java se accede a **la E/S estándar** a través de los atributos estáticos de la clase **java.lang.System**

Fields

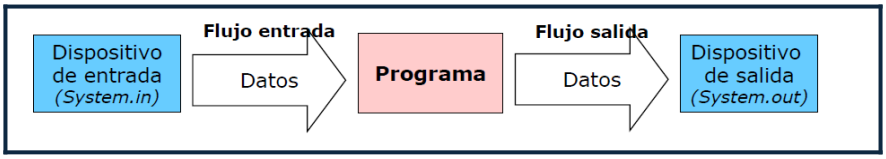
Modifier and Type	Field and Description
static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

System.in: implementa la entrada estándar. Por defecto es el **teclado**.

System.out: implementa la salida estándar. Por defecto es la **pantalla**.

System.err: implementa la salida de error. Por defecto es **la pantalla**

Estos flujos **standard el sistema se encarga de abrirlos y cerrarlos automáticamente**.



System.in:

- Instancia de la clase **InputStream**: **flujo de bytes** de entrada
- **Métodos**:
  - ✓ **read()**: permite leer **un byte** de la entrada como entero
  - ✓ **skip(n):** ignora n bytes de la entrada
  - ✓ **available():** número de bytes disponibles para leer en la entrada

System.out:

- Instancia de la clase **PrintStream** (subclase de OutputStream): **flujo de bytes** de salida
- **Métodos** para impresión de datos
  - ✓ **print(), println()**
  - ✓ **flush():** vacía el buffer de salida escribiendo su contenido

System.err

- Funcionamiento similar a System.out
- Se utiliza para enviar **mensajes de error** (por ejemplo a un fichero de log o a la consola)

3. Sistema de Ficheros y Directorios en Java

3.1 La clase FILE

En el paquete Java.io se encuentra la clase **FILE** para trabajar **directamente con los archivos y el sistema de archivos**.

Se utiliza para:

- ✓ **Obtener o modificar información asociada con un archivo**
- ✓ **Navegar por la jerarquía de subdirectorios**.

Un objeto **File representa un archivo o un directorio:**

> *Un **directorio en Java se trata igual que un archivo** con una propiedad adicional, una lista de nombres de archivo que se puede examinar utilizando el método LIST.*

Proporciona métodos de acceso a los archivos y operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

3.1.1 Creación de un objeto File

Los **constructores de File** permiten :

- ✓ **Inicializar el objeto** con el nombre de un archivo y la ruta donde se encuentra.

Constructores de la clase FILE	Ejemplos
`public File(String nombreCompleto)` Crea un objeto File con el nombre y ruta del archivo pasado como argumento	File f1= new File("/carpeta/archivo.txt"); File directorio=new File ("/");

`public File(String ruta, String nombre)` **Crea un objeto File en la ruta pasada como primer argumento y con el nombre del archivo como segundo argumento**	**File f2 = new File("/", "autoexec.bat");**
`public File(File ruta, String nombre)` **Crea un objeto File en la ruta que proporciona un objeto FILE pasada como primer argumento y con el nombre del archivo como segundo argumento.**	**File f3 = new File(directorio, "texto.txt");**
`public File(URI uri)` **Crea un objeto File a través de un objeto URI.**	**File f4= new File (new URI("file:///index.htm"));**

*Nota: Un **Uniform Resource Identifier o URI** (en español «identificador uniforme de recurso») es una cadena de caracteres corta que identifica inequívocamente un recurso (servicio, página, documento, dirección de correo electrónico, etc.). Normalmente estos recursos son accesibles en una red o sistema.*

3.1.2 Métodos de la clase File

Modificador y tipo	Método y descripción
`boolean`	`canExecute()` Devuelve true si el archivo existe y la aplicación puede ejecutarlo.
`boolean`	`canRead()` Devuelve true si el archivo existe y se puede leer.
`boolean`	`canWrite()` Devuelve true si el archivo existe y se puede escribir.
`int`	`compareTo(File ruta)` Compara dos rutas en orden alfabético
`boolean`	`createNewFile()` throws IOException Crea un nuevo archivo basado en la ruta dada al objeto File si solo si el fichero no existe. Hay que capturar la excepción IOException que ocurriría si hubo error crítico al crear el archivo. Devuelve true si se hizo la creación del archivo vacío y false si ya había otro archivo con ese nombre.
`static File`	`createTempFile(String prefijo, String sufijo)` throws IOException Crea un fichero vacío en el directorio temporal por defecto usando el prefijo y sufijo para generar el nombre. El prefijo y el sufijo deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo IllegalArgumentsException Requiere capturar la excepción IOException que se produce ante cualquier fallo en la creación del archivo
`static File`	`createTempFile(String prefijo, String sufijo, File directorio)` throws IOException Crea un fichero vacío en el directorio especificado usando el prefijo y sufijo para generar el nombre.
`boolean`	`delete()` Borra el fichero o directorio especificado al crear el objeto File. Devuelve true si puede hacerlo
`void`	`deleteOnExit()` Borra el fichero o directorio al finaliza la ejecución del programa.
`boolean`	`equals(Object objecto)` Devuelve true si la ruta especificada es igual que el objeto dado.
`boolean`	`exists()` Devuelve true si el fichero o directorio especificado existe.
`File`	`getAbsoluteFile()` Devuelve un objeto File con la ruta absoluta al objeto File creado.
`String`	`getAbsolutePath()` Devuelve una cadena con la ruta absoluta al objeto File.
`File`	`getCanonicalFile()` Convierte un objeto File a una única forma canónica más adecuada para las comparaciones.
`String`	`getCanonicalPath()` Convierte una ruta a una única forma canónica más adecuada para las comparaciones.
`long`	`getFreeSpace()` Devuelve el número de bytes libres en la partición.
`String`	`getName()` Devuelve el nombre del fichero o directorio especificado.
`String`	`getParent()` Devuelve una cadena con el directorio padre o `null` si no tiene un directorio padre.
`File`	`getParentFile()` Devuelve un objeto File con el directorio padre o `null` si no tiene un directorio padre..
`String`	`getPath()` Convierte la ruta especificada al crear el objeto File en una cadena
`long`	`getTotalSpace()` Devuelve el espacio total en la partición.
`long`	`getUsableSpace()` Devuelve el espacio utilizado por el fichero.
`int`	`hashCode()` Computes a hash code for this abstract pathname.

`boolean`	`isAbsolute()` Devuelve true si la ruta es absoluta.
`boolean`	`isDirectory()` Devuelve true si es un directorio.
`boolean`	`isFile()` Devuelve true si es un fichero.
`boolean`	`isHidden()` Devuelve true si el fichero es oculto.
`long`	`lastModified()` Devuelve la fecha de la última modificación del fichero especificado.
`long`	`length()` Devuelve la longitud del fichero especificado.
`String[]`	`list()` Devuelve un array de cadenas con los nombres y directorios del directorio especificado.
`String[]`	`list(FilenameFilter filtro)` Devuelve un array de cadenas con los nombres y directorios del directorio especificado que satisfacen el filtro.
`File[]`	`listFiles()` Lista los archivos que componen el directorio.
`File[]`	`listFiles(FileFilter filtro)` Lista los archivos que componen el directorio y que cumplen con el criterio especificado.
`File[]`	`listFiles(FilenameFilter filtro)` Lista los archivos que componen el directorio y que cumplen con el criterio especificado.
`Static File[]`	`listRoots()` Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
`Boolean`	`mkdir()` Crea un directorio en la ruta especificada.
`boolean`	`mkdirs()` Crea el directorio especificado por el objeto FILE aunque no exista el camino. Todos los directorios anteriores a la carpeta se crearán, si no existe. De este modo no saltará ninguna excepción si es que los directorios no existen.
`boolean`	`renameTo(File destino)` Renombra el fichero especificado
`boolean`	`setExecutable(boolean permiso, boolean soloPropietario)` Si `permiso` es true, se establece el permiso de ejecución.
`boolean`	`setExecutable(boolean permiso, boolean soloPropietario)` Si `permiso` es true, se establece el permiso de ejecución. Si `soloPropietario` es true, el permiso de ejecucion solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
`boolean`	`setLastModified(long fecha)` Establece el tiempo de la última modificación del archivo o directorio
`boolean`	`setReadable(boolean permiso)` Si `permiso` es true, se permiten operaciones de lectura. Devuelve true si la operación ha tenido éxito.
`boolean`	`setReadable(boolean permiso, boolean soloPropietario)` Si `permiso` es true, se permiten operaciones de lectura. Si `soloPropietario` es true, el permiso de lectura solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
`boolean`	`setReadOnly()` Establece el fichero o directorio de solo lectura.
`Boolean`	`setWritable(boolean permiso)` Si `permiso` es true, se permiten operaciones de lectura y escritura. Devuelve true si la operación ha tenido éxito.
`boolean`	`setWritable(boolean permiso, boolean soloPropietario)` Si `permiso` es true, se permiten operaciones de lectura y escritura. Si `soloPropietario` es true, el permiso de lectura y escritura solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
`Path`	`toPath()` Devuelve un objeto `java.nio.file.Path` a partir de la ruta especificada
`String`	`toString()` Devuelve una cadana a partir de la ruta especificada.
`URI`	`toURI()` Devuelve a `file:` URI que representa la ruta especificada.

Ejemplo:

```
import java.io.*;
import java.util.Date;
import java.text.SimpleDateFormat;
import java.util.Locale;
public class EjInformacionFichero {
 public static void main(String[] args) {
       String ruta=  "D:/ACCESO a DATOS";
// Para formatear la fecha de modificacion con el formato dd de mes de año en español
       SimpleDateFormat formateador =
           new SimpleDateFormat(   "dd 'de' MMMM 'de' yyyy", new Locale("es","ES"));
       File  f=new File(ruta);
        if(f.exists()){
            System.out.println("\n\tNombre: " + f.getName());
            System.out.println("\tTamaño: " + f.length()+ " bytes");
```

```
                System.out.println("\tRuta absoluta: " + f.getAbsolutePath());
                System.out.println("\tPuede leerse: " + f.canRead());
                System.out.println("\tPuede modificar: " + f.canWrite());
                System.out.println("\tEs archivo: " + f.isFile());
                System.out.println("\tEs oculto: " + f.isHidden());
                System.out.println("\tModificado por última vez: " +
                                   formateador.format(new Date(f.lastModified())));
                System.out.println("\tEs directorio: " + f.isDirectory());
                if(f.isDirectory()){
                    System.out.println("\n- ----Contenido del directorio: "+ruta+"- -------");
                    /* El  método list() devuelve un array con el contenido de un directorio */
                     String [] lista=f.list();
                     for(int i=0;i<lista.length;i++){
                            System.out.println(ruta+"/"+lista[i]);      }
                 }
             }
           else{   System.out.println("El "+ruta+" no existe");  }
    }    }
```

Un ejemplo de salida:

```
Nombre: ACCESO a DATOS
    Tamaño: 4096 bytes
    Ruta absoluta: D:\ACCESO a DATOS
    Puede leerse: true
    Puede editarse: true
    Es archivo regular: false
    Es oculto: false
    Modificado por última vez: 17 de abril de 2012
    Es directorio: true
    Es ejecutable: true
- --------Contenido del directorio: D:/ACCESO a DATOS- -------
D:/ACCESO a DATOS/Ejemplosiniciales
D:/ACCESO a DATOS/NETBEANS 7.1.1
D:/ACCESO a DATOS/MaterialAyuda
D:/ACCESO a DATOS/Unidades didácticas
BUILD SUCCESSFUL (total time: 0 seconds)
```

3.2 Java NIO

Java NIO (New I/O) ofrece una API más moderna y flexible para manejar archivos y directorios, con mejor rendimiento y opciones avanzadas de gestión de rutas, atributos y recorridos de árbol.

En Java, además de la clase **File** (más antigua, desde Java 1.0), existe desde **Java 7** el paquete **java.nio.file** que proporciona una forma más moderna, potente y flexible de trabajar con archivos y directorios. Facilita:

- Acceder a rutas y metadatos.
- Leer y escribir archivos.
- Manipular directorios.
- Copiar, mover o borrar archivos y directorios.
- Usar **streams** y **walks** para recorrer directorios.

3.2.1 Paquetes y clases principales de java.nio.file

Clase / Interfaz	Descripción	Métodos / Usos más destacados
Path	Representa **una ruta de archivo** o **directorio** (relativa o absoluta). Sustituye a File en muchos casos.	- **getFileName()** → nombre del archivo. - **getParent()** → directorio padre. -**toAbsolutePath()** → convierte en ruta absoluta. - **resolve(ruta)** → combina rutas. Se obtiene una nueva ruta que resulta de añadir el segundo Path al primero - **relativize(ruta)** → ruta relativa entre dos rutas.

		cuál es el camino desde una ruta de origen hasta una ruta de destino - **normalize()** → limpia las rutas eliminando referencias innecesarias como "." (que indica el directorio actual) o ".."
Paths	Clase de utilidades para **crear objetos Path**	- **get(String ruta)** → crea un Path a partir de una cadena. - **get(URI uri)** → crea un Path desde una URI.
Files	Contiene métodos estáticos para **trabajar con archivos y directorios**.	**-exists(path), isDirectory(path), size(path).** **-createFile(path),createDirectory(path), createDirectories(path).** **- copy(origen, destino, opciones), move(origen, destino, opciones), delete(path).** **-newDirectoryStream(path)** → listado. **-walk(path)** → recorrido recursivo. **-readAllLines(path)** → Lee todo el contenido de un archivo de texto **-write(path, datos)** →Escribe en el archivo las cadenas proporcionadas (sobrescribe si ya existe
FileSystem	Representa el **sistema de archivos**. Permite acceder a las raíces (C:, /, etc.)	- **FileSystems.getDefault()** → sistema de archivos actual. - **getRootDirectories**() → raíces del sistema. **-getPath(String ruta)** → crea un Path en este sistema.
DirectoryStream	Permite **recorrer el contenido de un directorio** de forma eficiente.	- **Iteración con for (Path p : stream**). **-Acepta filtros** (newDirectoryStream(path, "*.txt")).
FileVisitor<T>	Interfaz para **recorrer directorios recursivamente**.	- Métodos: **preVisitDirectory, visitFile, postVisitDirectory, visitFileFailed.** - Se usa con **Files.walkFileTree(path, visitor).**
SimpleFileVisitor<T>	Implementación base de **FileVisitor con métodos por defecto**. Se puede sobrescribir solo lo necesario.	Ejemplo: **sobrescribir visitFile() para listar archivos recursivamente**.
.attribute. BasicFileAttributes	Contiene **metadatos** de un archivo (tamaño, fechas, tipo, etc.).	- **size()** → tamaño en bytes. - **creationTime(), lastModifiedTime().** - **isDirectory(), isRegularFile().**
StandardCopyOption	Enum con opciones para copiar/mover archivos.	- **REPLACE_EXISTING** → sobrescribir si ya existe.

		- **COPY_ATTRIBUTES** → copiar metadatos. - **ATOMIC_MOVE** → movimiento atómico.

ANOTACION: Uso de / y \\ en rutas de Java y Windows

Separador en sistemas operativos:

Windows usa \ como separador de rutas (ej.: C:\Users\Pepa\archivo.txt).

Linux/macOS usan / (ej.: /home/pepa/archivo.txt).

En Java:

- Podemos **usar / en cualquier sistema**, incluso en Windows. Java lo interpreta y convierte internamente al separador correcto del sistema.

```
Path p = Paths.get("mi_carpeta/archivo.txt"); // Funciona en Windows y Linux
```

- **Si usas \ en cadenas Java, debes escaparlo (\\)** porque \ es carácter especial en strings (por ejemplo, \n, \t, etc..)

```
Path p = Paths.get("mi_carpeta\\archivo.txt"); // También válido
```

3.2.2 La Clase Path

Path es la base de NIO.

Representa una ruta, no un archivo real. Es como una dirección postal: puede existir o no, pero define una ubicación.

- **Creación**: Siempre se **crea a través de Paths.get()**. No puedes instanciarla directamente.

```
Path archivo = Paths.get("documento.txt");
```

Métodos Clave:

getFileName(): Devuelve el último elemento de la ruta (el nombre del archivo o directorio).

```
Path p = Paths.get("mi_carpeta/archivo.txt");
System.out.println(p.getFileName()); // Salida: archivo.txt
```

getParent(): Devuelve la ruta del directorio padre.

```
Path p = Paths.get("mi_carpeta/archivo.txt");
System.out.println(p.getParent()); // Salida: mi_carpeta
```

toAbsolutePath(): Convierte una ruta relativa en una absoluta.

```
Path p = Paths.get("archivo.txt");
System.out.println(p.toAbsolutePath());
// Salida: C:\Users\usuario\...\archivo.txt
```

resolve(Path other): Combina dos rutas. Es útil para construir rutas dinámicamente.

```
Path p1 = Paths.get("mi_carpeta");
Path p2 = p1.resolve("archivo.txt");
System.out.println(p2);
// Salida: mi_carpeta\archivo.txt
```

relativize(Path other): Calcula la ruta relativa desde esta Path hasta otra, es decir, **cómo llegar** desde una ruta base (origen) hasta otra (destino), usando pasos relativos (.. para subir directorios y nombres para bajar).

No devuelve una ruta absoluta, sino **el "camino" relativo** que tendrías que recorrer.

Ambas rutas deben ser del mismo tipo (ambas absolutas o ambas relativas), si no, relativize lanza IllegalArgumentException.

```
Path origen = Paths.get("/a/b/c");
Path destino = Paths.get("/a/x/y");
System.out.println(origen.relativize(destino));
// Salida: ../../x/y
```

.. (subir de c a b) ; .. (subir de b a a); x (bajar a x); y (bajar a y)

normalize(): Elimina elementos redundantes de una ruta, como . (directorio actual) o .. (directorio padre).

Devuelve un Path más limpio y directo

```
Path p = Paths.get("a/b/../c");
System.out.println(p.normalize());
// Salida: a\c
```

- Ruta original: a/b/../c
- b/.. significa: entra en b y luego sube al padre → se anula.
- Resultado: a/c

3.2.3 La Clase de Utilidad Paths

Paths no tiene estado; solo es una fábrica para crear objetos Path.

Métodos Clave:

get(String first, String... more): El método más utilizado.

Crea un Path a partir de uno o más fragmentos de cadena que representan partes de la ruta.de cadena.

```
Path ruta1 = Paths.get("C:", "Users", "admin");
Path ruta2 = Paths.get("C:/Users/admin"); // También funciona
```

get(URI uri): Crea un Path a partir de un URI (Uniform Resource Identifier).

Ventaja: Permite acceder a sistemas de archivos no convencionales, como:

Archivos ZIP/JAR tratados como directorios.

Sistemas de archivos remotos o virtuales (si hay un proveedor instalado).

```
URI uri = URI.create("jar:file:/C:/datos/miArchivo.zip");
try (FileSystem zipFs = FileSystems.newFileSystem(uri, Map.of())) {
    Path dentroZip = zipFs.getPath("/documento.txt");
    System.out.println(Files.readString(dentroZip));
}
// Al salir del bloque try, el FileSystem (ZIP) se cierra automáticamente.
//si el contenido de un archivo fuera binario
byte[] datos = Files.readAllBytes(dentroZip);
System.out.println("Tamaño del archivo: " + datos.length + " bytes");
```

- **URI:** Localiza el recurso (en este caso, un ZIP) usando un esquema especial (**jar:file:).**

- **FileSystems.newFileSystem(...)**: Monta el ZIP como si fuera un sistema de archivos, permitiendo navegarlo con Path y Files.
- **getPath(...):** Obtiene la ruta de un archivo dentro del ZIP.
- **Files.readString(...):** Lee el contenido del archivo directamente, sin necesidad de descomprimirlo manualmente. Intenta interpretar el contenido como texto usando una codificación (por defecto UTF-8).
- Ventaja: Puedes trabajar con el contenido de un ZIP igual que con carpetas normales, usando toda la API NIO.
    - Si el archivo no existe dentro del ZIP, el programa lanzará una excepción NoSuchFileException.
    - Si el ZIP no existe o la ruta es incorrecta, obtendrás una FileSystemNotFoundException o IOException.

3.2.4 La Clase de Utilidad Files

Files es el corazón de NIO.2. Contiene métodos estáticos para realizar **casi todas las operaciones con archivos y directorios**.

Ventaja: Permite **realizar operaciones** de forma **segura, atómica y multiplataforma**.

- **Segura: controla errores y permisos, evitando comportamientos inesperados**.

  Ejemplos:
    - Lanza excepciones claras (NoSuchFileException, AccessDeniedException) en lugar de fallar silenciosamente.
    - Comprueba permisos antes de leer o escribir.
    - Evita corrupción de datos al manejar flujos y buffers correctamente.
- **Atómica**: se realiza **completamente o no se realiza en absoluto**. No queda a medias aunque ocurra un fallo o se interrumpa el proceso.

  Ejemplo:

  ```
  Files.move(origen, destino, StandardCopyOption.ATOMIC_MOVE);
  ```

  Si el movimiento falla, el archivo original sigue intacto.

  No existe un estado intermedio en el que el archivo esté "a medias" en ambos sitios.

  Esto es clave para evitar inconsistencias en sistemas críticos (bases de datos, backups, etc.).
- **Multiplataforma**: el mismo código **funciona en distintos sistemas operativos** gracias a la abstracción de Java , es decir, funciona en Windows, Linux y macOS sin cambios.

  Java abstrae las diferencias:
    - Convierte automáticamente separadores de ruta (/ o \).
    - Usa el FileSystem por defecto del sistema.
    - Adapta internamente permisos y atributos según el SO.

Métodos para Verificación y Metadatos:

Método	Descripción	Ejemplo
Files.exists(path)	Comprueba si la ruta existe.	Files.exists(Paths.get("datos.txt"))
Files.isDirectory(path)	Devuelve true si es un directorio.	Files.isDirectory(Paths.get("miCarpeta"))
Files.isRegularFile(path)	Devuelve true si es un archivo normal (no directorio)	Files.isRegularFile(Paths.get("datos.txt"))

Files.size(path)	Devuelve el tamaño en bytes.	Files.size(Paths.get("datos.txt"))

Ejemplo:

```
Path ruta = Paths.get("datos.txt");
if (Files.exists(ruta)) {
    System.out.println("Tamaño: " + Files.size(ruta) + " bytes");
    if (Files.isRegularFile(ruta)) {
        System.out.println("Es un archivo normal");
    }}
```

Métodos para Creación

Método	Descripción	Ejemplo
Files.createFile(path)	Crea un archivo vacío. Falla si ya existe.	Files.createFile(Paths.get("nuevo.txt"))
Files.createDirectory(path)	Crea un único directorio.	Files.createDirectory(Paths.get("nuevaCarpeta"))
Files.createDirectories(path)	Crea un directorio y todos sus padres si no existen.	Files.createDirectories(Paths.get("padre/hijo/nieto"))

Ejemplo:

```
Path dir = Paths.get("proyectos/java");
Files.createDirectories(dir); // Crea toda la estructura si no existe
Path archivo = dir.resolve("main.java");//combina la ruta de forma
                                        //segura y multiplataforma
Files.createFile(archivo); // Crea el archivo vacío
```

Métodos para Manipulación

Método	Descripción	Ejemplo
Files.copy(origen, destino, opciones...)	Copia un archivo o directorio.	Files.copy(src, dest, StandardCopyOption.REPLACE_EXISTING)
Files.move(origen, destino, opciones...)	Mueve o renombra.	Files.move(src, dest, StandardCopyOption.ATOMIC_MOVE)
Files.delete(path)	Borra un archivo o directorio vacío.	Files.delete(Paths.get("archivo.txt"))

StandardCopyOption

Este enum (una constante pública, estática y final con un cojunto posible de valores válidos) **define opciones para las operaciones de copia y movimiento**.

- **Opciones Comunes**:
  - **REPLACE_EXISTING**: Sobrescribe el archivo de destino si ya existe.
  - **COPY_ATTRIBUTES**: Copia los metadatos (fechas, permisos, etc.) del archivo de origen.
  - **ATOMIC_MOVE**: Realiza el movimiento como una operación atómica. Si falla, el archivo original permanece inalterado.
- **Uso**: Se pasan como argumentos a los métodos **Files.copy() y Files.move().**

Ejemplo:

```
Path origen = Paths.get("datos.txt");
Path copia = Paths.get("backup/datos.txt");
Files.createDirectories(copia.getParent());
Files.copy(origen, copia, StandardCopyOption.REPLACE_EXISTING);
```

Métodos para Lectura y Escritura

Método	Descripción	Ejemplo
`Files.readAllLines (path)`	**Lee todas las líneas de un archivo de texto y las devuelve en una `List<String>`.** Útil para archivos de texto pequeños.	`List<String> lineas = Files.readAllLines (Paths.get("notas.txt")); for (String linea : lineas) { System.out.println(linea);}`
`Files.write (path, data)`	**Escribe datos en un archivo. Si existe, sobrescribe por defecto**. `data` puede ser una `List<String>` o un `byte[]`.	`Files.write (Paths.get("notas.txt"), List.of("Línea 1", "Línea 2"));`
`Files.readAllBytes (path)`	**Lee todo el contenido de un archivo y lo devuelve como un `byte[]`**. Ideal para archivos binarios (imágenes, PDF, ZIP…).	`byte[] datos = Files.readAllBytes (Paths.get("imagen.png")); System.out.println("Tamaño: " + datos.length + " bytes")`

Métodos para Listado y Recorrido

Método	Descripción	Ejemplo
Files.list(path)	**Lista el contenido inmediato de un directorio** (no recursivo) devolviendo un **`Stream<Path>`.** Ideal para usar con programación funcional (`filter`, `map`, etc.).	Recorre cada Path del Stream y lo imprime.Mostrará la ruta completa o relativa según cómo se haya creado el Path try (Stream<Path> s = Files.list(Paths.get("documentos"))) {s.forEach(System.out::println);} //está **utilizando programación funcional** **De manera clásica:** Path dir = Paths.get("documentos"); // try-with-resources para cerrar automáticamente el Stream try (Stream<Path> stream = Files.list(dir)) { Iterator<Path> it = stream.iterator(); while (it.hasNext()) { Path p = it.next(); System.out.println(p); } } catch (IOException e) { e.printStackTrace(); } }

Files.newDirectoryStream (path)	**Itera sobre el contenido de un directorio** (no recursivo). Devuelve un `DirectoryStream<Path>`	Path dir = **Paths.get("documentos");** try (**DirectoryStream<Path> stream = Files.newDirectoryStream(dir))** **//para recorrer** { for (Path p : stream) { System.out.println (p.getFileName()); } }
Files.walk (path)	**Recorre recursivamente un árbol de directorios devolviendo un** `Stream<Path>.`	try (**Stream<Path>** stream = **Files.walk(Paths.get("miCarpeta")))** {//para recorrerlo **Iterator<Path> it = stream.iterator();** while (**it.hasNext()**) { Path p = it.next(); System.out.println(p); } } catch (IOException e) { e.printStackTrace(); } Ejemplo de salida: miCarpeta miCarpeta\archivo1.txt miCarpeta\archivo2.txt miCarpeta\subcarpeta1 miCarpeta\subcarpeta1\foto.png

3.2.5 Tipos de recorrido de directorios: Stream<Path> y DirectoryStream<Path>

Cuando usamos métodos de la **clase Files para listar o recorrer directorios** (*Files.list, Files.walk, Files.newDirectoryStream…*), **no obtenemos directamente una lista o array**, sino **objetos especiales** que **representan un flujo de rutas (Path)** que podemos procesar. Estos objetos pueden ser:

- **Stream<Path>**
- **DirectoryStream<Path>**

Stream<Path>

- **Qué es**: **Un flujo (stream) de elementos Path** de la API de Streams de Java 8+, es decir, una secuencias de rutas(Path).

  Cada **elemento es un objeto Path** que representa un archivo o directorio en el sistema de archivos.
- **Quién lo devuelve:**

  Files.list(Path) → lista el contenido inmediato (no recursivo).

  Files.walk(Path) → recorre recursivamente.
- **Características:**
  - Permite usar **directamente operaciones funcionales** (filter, map, sorted, collect, etc.).
  - Procesa **los elementos de forma perezosa (lazy),** sin cargarlos todos en memoria.
  - Se **cierra automáticamente con try-with-resources**.
  - Un Stream<Path> **no es una colección reutilizable**: una vez ejecutas una operación terminal, el **stream se cierra y no se puede volver a recorrer**. Si necesitas guardar los resultados, usa .collect(Collectors.toList()) o .toList().

Ejemplo:

```
try (Stream<Path> s = Files.walk(Paths.get("miCarpeta")))
 {
    s.filter(Files::isRegularFile)    // Intermedia: solo archivos
     .map(Path::getFileName)          // Intermedia: solo el nombre
     .sorted()                        // Intermedia: orden alfabético
     .forEach(System.out::println);   // Terminal: imprime
}
//aquí se cierra y no se podría utilizar de nuevo
```

Ejemplo de almacenar en una lista los streams

```
Path dir = Paths.get("miCarpeta");
List<Path> archivos;
try (Stream<Path> s = Files.walk(dir)) {
    archivos = s.filter(Files::isRegularFile)
                .map(Path::getFileName)
                .sorted()
                .collect(Collectors.toList()); // Guardar en lista
}
// Ahora puedes usar la lista varias veces
//imprimir con programacion funcional
archivos.forEach(System.out::println); // Primera vez
System.out.println("Total: " + archivos.size()); // Segunda vez

//imprimir de forma clásica
// Primera vez: recorrer e imprimir
for (Path p : archivos) {
    System.out.println(p);
}
// Segunda vez: mostrar el total
System.out.println("Total: " + archivos.size());
```

1. filter(...) — Filtrar elementos del stream

Recibe una función que devuelve true o false para cada elemento.

En Stream<Path> se usa para **quedarse solo con las rutas que cumplan una condición**.

Ejemplo	Qué hace
.filter(Files::isRegularFile)	Solo archivos (no carpetas).
.filter(Files::isDirectory)	Solo directorios.
.filter(p -> p.toString().endsWith(".txt"))	Solo archivos .txt.
.filter(p -> p.getFileName(). toString().startsWith("doc_"))	Solo nombres que empiecen por doc_.
.filter(p -> { try { return Files.size(p) > 1024; } catch (IOException e) { return false; } })	Solo archivos de más de 1 KB.

2. map(...) — Transformar cada elemento

Transforma cada elemento en otro valor.

En Stream<Path> se usa para **extraer información** o **convertir el tipo**.

Ejemplo	Qué devuelve
`.map(Path::getFileName)`	Solo el nombre del archivo (Path).
`.map(Path::toString)`	Ruta como String.
`.map(Path::toAbsolutePath)`	Ruta absoluta (Path).
`.map(Path::getParent)`	Carpeta padre (Path).
`.map(p -> p.getName(0))`	Primer componente de la ruta (Path).
`.map(p -> { try { return Files.size(p); } catch (IOException e) { return -1L; } })`	Tamaño en bytes (Long).
`.map(p -> { try { return Files.getLastModifiedTime(p); } catch (IOException e) { return null; } })`	Fecha de última modificación (FileTime).
`.map(p -> p.getFileName().toString().toUpperCase())`	Nombre en mayúsculas (String).

3. sorted(...) — Ordenar elementos

Ordena el stream. En Stream<Path> se basa en el **orden natural** de Path (alfabético por ruta), pero se puede pasar un Comparator para personalizarlo.

Cómo funciona la comparación por defecto:

- Path implementa Comparable<Path> y su compareTo compara cada componente de la ruta en orden.
- Si son rutas absolutas, la comparación empieza por la raíz (C:\, /home, etc.) y sigue por cada subdirectorio.
- Esto significa que el orden depende de todo el texto de la ruta, no solo del nombre del archivo.

Ejemplo: Tenemos estas rutas

/home/pepa/a.txt

/home/pepa/b.txt

/home/zeta/a.txt

Resultado:

Orden natural (sorted()):

1. /home/pepa/a.txt
2. /home/pepa/b.txt
3. /home/zeta/a.txt

Ejemplo	Orden
`.sorted()`	Alfabético ascendente por ruta completa.
`.sorted(Comparator.reverseOrder())`	Alfabético descendente por ruta completa.
`.sorted(Comparator.comparing(Path::getFileName))`	Ascendente por nombre de archivo.
`.sorted(Comparator.comparing(p -> p.getFileName().toString().toLowerCase()))`	Ascendente por nombre ignorando mayúsculas/minúsculas.
`.sorted(Comparator.comparingLong(p -> { try { return Files.size(p); } catch (IOException e) { return 0L; } }))`	Ascendente por tamaño.
`.sorted(Comparator.comparingLong(p -> { try { return Files.size(p); }`	Descendente por tamaño.

catch (IOException e) { return 0L; } }).reversed())	
.sorted(Comparator.comparing(p -> { try { return Files.getLastModifiedTime(p); } catch (IOException e) { return FileTime.fromMillis(0); } }))	Ascendente por fecha de modificación.

- **Ejemplo combinando todo:**

  Recorre recursivamente miCarpeta y sus subcarpetas, selecciona solo los archivos .txt, los convierte a ruta absoluta, los ordena por nombre ignorando mayúsculas y los imprime.

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.FileTime;
import java.util.Comparator;
import java.util.stream.Stream;

public class EjemploStreamPath {
    public static void main(String[] args) {
        Path dir = Paths.get("miCarpeta");

        try (Stream<Path> s = Files.walk(dir)) {
            s.filter(Files::isRegularFile) // Solo archivos
             .filter(p -> p.toString().endsWith(".txt")) // Solo .txt
             .map(Path::toAbsolutePath) // Ruta absoluta
             .sorted(Comparator.comparing(p ->
              p.getFileName().toString().toLowerCase())) // Orden por
                                  //nombre ignorando mayúsculas
             .forEach(System.out::println); // Imprime
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}


//nota para ordenarlo por nombre en orden inverso
.sorted(Comparator.comparing(
    p -> p.getFileName().toString().toLowerCase(),
    Comparator.reverseOrder()
))
```

DirectoryStream<Path>

Es una interfaz de **Java NIO.2** que permite **recorrer de forma eficiente el contenido de un directorio**.

A diferencia de Files.list() o Files.walk(), **no devuelve un Stream<Path>,** sino un **objeto iterable (Iterable<Path>)**, pensado para usarse con un bucle **for-each clásico**.

Se crea con **Files.newDirectoryStream().**

Características principales

- **Recorrido inmediato**: lista solo el contenido **inmediato** del directorio (no recursivo).
- **Eficiencia**: muy rápido y con bajo consumo de memoria, ideal para directorios grandes.
- **Filtrado previo**: permite aplicar un patrón ***glob*** **al crearlo**, para que solo devuelva lo que coincide.
- **Cierre de recursos**: debe usarse con *try-with-resources* para liberar el descriptor de directorio.
- **Sin API funcional directa**: no tiene map, filter, sorted… salvo que lo conviertas a Stream con StreamSupport.

Ejemplo:

```
Path dir = Paths.get("archivos");

try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir)) {
    for (Path p : ds) {
        System.out.println(p.getFileName());
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Un **patrón glob es una cadena que describe un conjunto de nombres de archivo usando comodines**. Se usa para filtrar rutas por nombre o extensión de forma sencilla, similar a la terminal de Linux o Windows.

Símbolo	Significado	Ejemplo	Coincide con
*	Cualquier secuencia de caracteres (0 o más)	"*.txt"	notas.txt, a.txt
?	Un solo carácter cualquiera	"foto?.jpg"	foto1.jpg, fotoA.jpg
{a,b}	Alternativa: coincide con a o con b	"*.{jpg,png}"	imagen.jpg, logo.png
[abc]	Un solo carácter de la lista	"file[123].txt"	file1.txt, file2.txt
[a-z]	Un solo carácter en el rango	"letra[a-z].txt"	letraa.txt, letrab.txt

Ejemplo

```
Path dir = Paths.get("archivos");

try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "*.txt"))
   {
    for (Path p : ds) {
        System.out.println(p.getFileName()); // Solo .txt
    }
}
```

Conversión a Stream<Path> para usar API funcional

- DirectoryStream<Path> **no** es un Stream, sino un Iterable<Path>.
- La API funcional de Java (Streams) **solo funciona** con objetos de tipo Stream<T>.
- Para convertirlo a un Stream, Java nos da la clase StreamSupport y la interfaz Spliterator.

Cómo se hace la conversión

1. Obtener el **Spliterator** del DirectoryStream Como DirectoryStream es Iterable, tiene el método .spliterator().
2. Usar **StreamSupport.stream**(...) Este método recibe:
   - El Spliterator obtenido.
   - Un booleano parallel (true para stream paralelo, false para secuencial).

Si quieres usar filter, map, sorted… puedes convertirlo así:

```
import java.util.stream.StreamSupport;
Path dir = Paths.get("archivos");
try (DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "*.txt"))
   {
    StreamSupport.stream(ds.spliterator(), false)
                 .map(Path::getFileName)
                 .sorted()
                 .forEach(System.out::println);
}
```

3.2.6 FileSystem

FileSystem es una abstracción que representa el sistema de archivos de tu sistema operativo.

Método	Descripción	Ejemplo
FileSystems.getDefault()	Obtiene el **FileSystem predeterminado**, que representa el sistema de archivos de tu máquina. Es un objeto que sabe cómo acceder, interpretar y manipular rutas, archivos y directorios **en el sistema operativo donde se está ejecutando tu programa**	FileSystem fs = **FileSystems.getDefault();** System.out.println("Separador: " + fs.getSeparator());
getRootDirectories()	Devuelve un **Iterable<Path> con las raíces del sistema de archivos** (por ejemplo, C:\, D:\ en Windows o / en Linux).	FileSystem fs = **FileSystems.getDefault();** for (Path root : **fs.getRootDirectories())** {System.out.println(root);}
getPath(String first, String... more)	**Crea un Path vinculado a este FileSystem** a partir de uno o más fragmentos de cadena.	FileSystem fs = FileSystems.getDefault(); Path p = fs.getPath("carpeta", "archivo.txt"); System.out.println(p);

3.2.7 FileVisitor<T> y SimpleFileVisitor<T>: Recorrido recursivo

- **Files.walkFileTree(...) es un método estático** de la clase java.nio.file.Files (no de File de la API antigua, sino de la clase moderna de NIO.2) .

  Sirve para **recorrer recursivamente un árbol de directorios** ejecutando acciones personalizadas en cada carpeta y archivo.

  - ✓ **Función:** Recorre un **directorio y todos sus subdirectorios** (un *árbol de directorios*) llamando a los métodos de un **visitante (FileVisitor<Path>).**
  - ✓ **Estilo:** Imperativo, **basado en *callbacks*** (métodos que tú defines y que Java invoca en cada paso del recorrido).
  - ✓ **Ventaja:** Control total sobre el recorrido: puedes decidir entrar o no en ciertas carpetas, parar en cualquier momento, manejar errores, etc.

- **FileVisitor<Path>:** Es una interfaz genérica de java.nio.file que **define cómo recorrer un árbol de directorios de forma recursiva.**
  - ✓ Se usa junto con el método `Files.walkFileTree(...)`.
  - ✓ El tipo genérico `<T>` suele ser `Path` → `FileVisitor<Path>`.

- **Métodos que defiene:** Llama a cuatro métodos en tu **FileVisitor<Path>:**
  - **preVisitDirectory(dir, attrs)** → antes de entrar en un directorio.
  - **visitFile(file, attrs)** → al encontrar un archivo.
  - **visitFileFailed(file, exc)** → si falla el acceso a un archivo.
  - **postVisitDirectory(dir, exc)** → al salir de un directorio (sea exitoso o con error).

Cada método devuelve un **FileVisitResult**:

- CONTINUE → seguir normalmente.
- SKIP_SUBTREE → saltar el contenido del directorio actual.
- SKIP_SIBLINGS → saltar el resto de elementos en el mismo nivel.
- TERMINATE → detener el recorrido.

- **SimpleFileVisitor<Path>**

  Permite sobrescribir solo los métodos que necesites, sin tener que implementar los cuatro.

  Ventajas
  - Código más limpio y corto.
  - Ideal para tareas simples o cuando solo necesitas uno o dos métodos.
  - Muy usado para recorridos recursivos con Files.walkFileTree(...).

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

public class EjemploVisitor {
    public static void main(String[] args) throws IOException {
        // Ruta inicial desde la que comenzará el recorrido
        Path inicio = Paths.get("miCarpeta");

        // Recorre recursivamente el árbol de directorios
        Files.walkFileTree(inicio, new SimpleFileVisitor<Path>() {

            // Método que se ejecuta al visitar cada archivo
        @Override
          public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
   {
                // Si el archivo termina en .txt, lo mostramos
                if (file.toString().endsWith(".txt")) {
                    System.out.println("Archivo TXT: " + file);
                }
               // CONTINUE indica que siga con el siguiente archivo o directorio
                return FileVisitResult.CONTINUE;
            }

            // Método que se ejecuta antes de entrar en un directorio
            @Override
          public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes
   attrs) {
                System.out.println("Entrando en: " + dir);
                return FileVisitResult.CONTINUE;
            }
```

```
        });  }}
```

Diferencia con Files.walk(...) **y** Files.walkFileTree(...)

- Files.walk(...) → Más conciso, ideal para procesar datos en cadena con filter, map, sorted…
- Files.walkFileTree(...) → Más control sobre el recorrido (puedes saltar carpetas, parar, manejar errores de forma más fina).

3.2.8 BasicFileAttributes

- Es una **interfaz** de **java.nio.file.attribute** que representa un conjunto de **atributos básicos** de un archivo o directorio.
- Contiene información como:
  - Si es archivo regular, directorio .
  - Tamaño en bytes.
  - Fechas de creación, última modificación y último acceso.

◆ **Dónde se utiliza**

1. **En FileVisitor<Path> y SimpleFileVisitor<Path>**
   - Los métodos como visitFile(...) y preVisitDirectory(...) reciben un parámetro BasicFileAttributes attrs con los atributos del archivo o directorio que se está visitando.

```
@Override
public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
    if (attrs.isRegularFile()) {
            System.out.println("Archivo: " + file + " Tamaño: " +
   attrs.size());
    }
    return FileVisitResult.CONTINUE;
```

Con Files.readAttributes(...)

- Puedes obtener los atributos de un archivo sin recorrer un árbol de directorios.

```
BasicFileAttributes attrs = Files.readAttributes(ruta,
   BasicFileAttributes.class);
System.out.println("Tamaño: " + attrs.size());
System.out.println("Última modificación: " + attrs.lastModifiedTime());
```

2. **En comprobaciones rápidas**
   - Métodos como attrs.isDirectory(), attrs.isRegularFile(), attrs.isSymbolicLink() permiten saber el tipo de entrada.

◆ **Métodos más usados de BasicFileAttributes**

Método	Devuelve	Uso
isRegularFile()	boolean	Si es un archivo normal.
isDirectory()	boolean	Si es un directorio.
isSymbolicLink()	boolean	Si es un enlace simbólico.
size()	long	Tamaño en bytes.
creationTime()	FileTime	Fecha/hora de creación.
lastModifiedTime()	FileTime	Fecha/hora de última modificación.
lastAccessTime()	FileTime	Fecha/hora de último acceso.

4. Manejo de archivos en Java: lectura y escritura de datos

Para el manejo de archivos, la **API de Entrada/Salida (IO)** de Java proporciona diferentes clases especializadas, agrupadas en las siguientes cuatro categorías según el tipo de contenido que gestionan:

Tipo de archivo	Lectura (Input)	Escritura (Output)	Modo de acceso
Binarios (bytes)	FileInputStream, BufferedInputStream, DataInputStream	FileOutputStream, BufferedOutputStream, DataOutputStream	**Secuencial**
Texto (caracteres)	FileReader, BufferedReader, InputStreamReader	FileWriter, BufferedWriter, PrintWriter	**Secuencial**
Objetos	ObjectInputStream	ObjectOutputStream	**Secuencial**
Acceso aleatorio	RandomAccessFile	RandomAccessFile	**Aleatorio**

4.1 Apertura y cierre de ficheros en Java

Antes de trabajar con las clases específicas de entrada y salida, es fundamental entender cómo se abren y cierran los ficheros en Java y qué ocurre en cada caso.

Apertura de ficheros

Cuando abrimos un fichero en Java para lectura o escritura, se establece una conexión entre el programa y el sistema de archivos.

- Lectura: **si el fichero no existe, se lanza una excepción (FileNotFoundException).**
- Escritura:
  - Si el fichero no existe, **se crea automáticamente**.
  - Si el fichero ya existe, por defecto **se sobrescribe y** el puntero de escritura se coloca al principio del archivo.
  - Si queremos conservar el contenido existente y añadir datos al final, debemos **abrirlo en modo append** (new FileOutputStream("archivo.dat", true)).

Cierre de ficheros

Siempre que se **termina de trabajar con un fichero, hay que cerrar el flujo para liberar recursos del sistema**.

Si no se cierra, pueden producirse fugas de memoria, bloqueos o pérdida de datos no escritos en el disco.

Formas de cerrar un fichero

- **Forma clásica (try–catch–finally)**

  Antes de Java 7, se usaba un bloque finally para garantizar el cierre:

```
FileInputStream fis = null;
try {
    fis = new FileInputStream("datos.bin");

} catch (IOException e) {

} finally {
    if (fis != null) {
        try {
            fis.close(); // cierre manual
```

```
            } catch (IOException e) {         }     }}
```

- **2. Forma moderna: try-with-resources**

  Desde Java 7, se recomienda usar **try-with-resources**, que cierra automáticamente los flujos al salir del bloque:

```
// El flujo se declara dentro del paréntesis del try
try (FileOutputStream fos = new FileOutputStream("salida.txt", true)) {
    fos.write("Hola mundo\n".getBytes());
    System.out.println("Texto escrito en modo append.");
} catch (IOException e) {
    e.printStackTrace();
}
// No hace falta cerrar: se cierra solo
```

🔑 **Ventajas de try-with-resources**

- Código más limpio y legible.
- Evita olvidarse de cerrar el fichero.
- Cierra automáticamente incluso si ocurre una excepción.
- Compatible con todas las clases que implementan AutoCloseable (como FileInputStream, FileOutputStream, BufferedReader, etc.).

4.2 Clases FileOutputStream y FileInputStream

FileOutputStream y FileInputStream son clases que manipulan **ficheros**. Son herederas de **Input/OutputStream**, por lo que manejan corrientes de datos en forma de **bytes** binarios.

- La clase **FileInputStream (hereda de InputStream):**
  - ✓ **Acceso a datos almacenados en ficheros** sin demasiadas pretensiones de formateo, y con **acceso secuencial.**

Constructores de la clase FileInputStream	Ejemplos
`FileInputStream (File fichero)` Crea un flujo de entrada FileInputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto *File fichero.* Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo	File miFichero = new File("texto.txt"); FileInputStream miFicheroSt = new FileInputStream(miFichero);
`FileInputStream(String nombreFichero)` Crea un flujo de entrada FileInputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena *nombreFichero.* Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo	FileInputStream miFicherot = new FileInputStream("C:\texto.txt");
`FileInputStream(FileDescriptor DescriptorFichero )` Crea un flujo de entrada FileInputStream mediante el descriptor de Fichero.	FileInputStream Fichero1= new FileInputStream(); FileDescripotr fd = Fichero1.getFD(); FileInputStream Fichero2 = new FileInputStream(fd);

Metodos de la clase FileInputStream

Modificador y tipo	Method and Description
int	**available**() Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	**close**() Cierra el fichero y libera los recursos del sistema que esté usando..
FileChannel	**getChannel**() Retorna el objecto **FileChannel** asociado con el fichero. Un FileChannel es una clase de la librería Java NIO (java.-nio.channels.FileChannel) que representa un canal de acceso a un archivo
FileDescriptor	**getFD**() Devuelve el objecto FileDescriptor que representa la conexion actual del fichero en el sistema de ficheros usado por el FileInputStream.
int	**read()** Lee un byte de datos desde el fichero. Devuelve -1 si no hay ningún byte más que leer.
int	**read**(byte[] b) No lee un solo byte, sino que lee hasta que b.length bytes guardándolos en el array b que se envía como paráme-tro. Devuelve -1 si no hay ningún byte más que leer.
int	**read**(byte[] b, int off, int len) Lee hasta len bytes del fichero y los deposita en b a partir de off. . Devuelve -1 si no hay ningún byte más que leer.

long	**skip**(long n) Salta n bytes de datos del fichero.

- La clase **FileOutputStream (hereda de OutputStream):**
  - ✓ **Almacenar datos en un fichero secuencial**.

Constructores de la clase FileOutputStream	Ejemplos
`FileOutputStream (File fichero)` Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto *File fichero*. Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo. Si el fichero existe, los datos que contienen se borrarán.	File miFichero = new File("texto.txt"); FileOutputStream miFicheroSt = new FileOutputStream(miFichero);
`FileOutputStream (File fichero, boolean agregar)` Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto *File fichero*. Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo. Si el fichero existe y el parámetro agregar es cierto, entonces los bytes se escriben en el final del archivo y no el principio, es decir, no se pierde la información del fichero.	File miFichero = new File("texto.txt"); FileOutputStream miFicheroSt = new FileOutputStream(miFichero,true);
`FileOutputStream (String nombreFichero)` Crea un flujo de salida FileIOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena *nombreFichero*. Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo	FileOutputStream miFicherot = new FileOutputStream("C:\texto.txt");
`FileOutputStream (String nombreFichero, boolean agregar)` Crea un flujo de salida FileIOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena *nombreFichero*. Un objeto *FileDescriptor* se crea para representar a esta conexión de archivo Si el fichero existe y el parámetro agregar es cierto, entonces los bytes se escriben en el final del archivo y no el principio, es decir, no se pierde la información del fichero.	FileOutputStream miFicherot = new FileOutputStream("C:\texto.txt",true);
`FileOutputStream(FileDescriptor DescriptorFichero )` Crea un flujo de entrada FileOutputStream mediante el descriptor de Fichero.	FileIOutputStream Fichero1= new FileOutputStream(); FileDescripotr fd = Fichero1.getFD(); FileOutputStream Fichero2 = new FileOutputStream(fd);

Metodos de FIileOutputStream

Modificador y tipo	Métodos y descripición
void	**close**() Cierra el flujo de salida y libera todos los recursos del sistema asociados con esta corriente. Cualquier acceso posterior generaría una IOException
FileChannel	**getChannel**() Devuelve el objeto FileChannel único asociado a este flujo de salida del archivo.
FileDescriptor	**getFD**() Devuelve el descriptor de fichero asociado con el flujo de salida.
void	**write(int b)** Escribe el byte especificado a en el flujo de salida del archivo.
void	**write**(byte[] b) Escribe todo el array de bytes en la corriente de salida
void	**write**(byte[] b, int posinicial, int numbytes) Escribe el array de bytes en el flujo de salida, pero empezando por la posición inicial y sólo la cantidad indicada por numbytes.

Excepciones: FileOutputStream y FileInputStream pueden lanzar la excepción:

- **FileNotFoundException**: Se lanza si el archivo no existe, si es un directorio en lugar de un archivo normal, o por alguna otra razón no se puede abrir para la lectura o para la escritura.

Ej: Lectura secuencial **byte a byte** de un archivo de texto. Cierre del fichero utilizando try-with-resources

```
public class EjFicheroSecuencialBytes {
    public static void main(String[] args) {
                // Creamos un objeto File que representa la ruta del archivo
        File f = new File("C:/FicherosJava/ejemplo.txt");

        // try-with-resources: el flujo se abre aquí y se cerrará automáticamente
        // al salir del bloque try, incluso si ocurre una excepción
        try (FileInputStream fis = new FileInputStream(f)) {
            int datos; // variable para almacenar cada byte leído
              // read() devuelve un entero entre 0 y 255 (el byte leído)
            // o -1 si se ha llegado al final del archivo
            while ((datos = fis.read()) != -1) {
                // Convertimos el byte leído a carácter para mostrarlo en pantalla
                System.out.print((char) datos);
```

```
            }
        } catch (FileNotFoundException e) {
            // Se lanza si el archivo no existe en la ruta indicada
            System.out.println("El fichero " + f.getName() + " no se encuentra");
        } catch (IOException e) {
            // Se lanza si ocurre un error de entrada/salida durante la lectura
            System.out.println("Error en lectura de datos");
        }    }}
```

Ejemplo: Escritura de forma secuencial en un archivo. Cierre en **finally** con método close

```
public class EjEscrituraSecuencialBytes {
    public static void main(String[] args) {
        int contLin = 0; // contador de líneas escritas
        String lineas[] = {"primera linea","segunda linea",
            "tercera linea",  "cuarta linea"      };
        byte[] s;
        FileOutputStream f = null; // flujo de salida
        try {
         // Abrimos el archivo para escritura (si no existe se crea, si existe se sobrescribe)
            f = new FileOutputStream("C:/FicherosJava/ejemplo1.txt");

            // Escribimos cada línea en el archivo
            for (int i = 0; i < lineas.length; i++) {
                // Convertimos la cadena en un array de bytes
                s = lineas[i].getBytes();
                f.write(s); // grabamos el array de bytes
                f.write((byte) '\n'); // salto de línea
                contLin++; // incrementamos el contador
            }
            System.out.println("Grabadas " + contLin + " lineas (éxito)");
        } catch (IOException e) {
            System.out.println("Problema en la grabación");
        } finally {
            // Cierre tradicional: se hace en finally para asegurar que se ejecute siempre
            if (f != null) {
                try {
                    f.close();
                } catch (IOException e) {
                    System.out.println("Error al cerrar el fichero de escritura");
                }
            }
        }}
```

4.3 Clases FileReader y FileWriter

FileReader y **FileWriter** son clases que manipulan **ficheros de texto**. Son herederas de **Reader/Writer**, por lo que manejan corrientes de datos en forma de **caracteres** en lugar de bytes.

En Java, además de las clases que trabajan con bytes (FileInputStream, FileOutputStream), existen clases que trabajan directamente con **caracteres (FileWriter y FileReader)**.

Esto es **más cómodo cuando se manejan archivos de texto**, ya que se encargan de la conversión entre la codificación del sistema y el Unicode interno de Java**.**

Características generales

- **FileReader**: clase para leer caracteres de un archivo. Es el equivalente a FileInputStream, pero orientado a caracteres.
- **FileWriter**: clase para escribir caracteres en un archivo. Es el equivalente a FileOutputStream, pero orientado a caracteres.
- Se pueden combinar con **BufferedReader y BufferedWriter** para mejorar el rendimiento y añadir métodos útiles **como readLine() o newLine().**
- También se puede **usar InputStreamReader y OutputStreamWriter** para **convertir flujos de bytes en flujos de caracteres**, especificando la codificación (ej. UTF-8).

FILEWRITER

Constructor	Descripción	Ejemplo
Modificador y método	Descripción	
void close()	Cierra el flujo de salida y libera todos los recursos del sistema asociados. Cualquier intento posterior de escritura generará una IOException.	
void write(int c)	Escribe un único carácter (código Unicode) en el archivo.	
void write(char[] cbuf)	Escribe todo el contenido de un array de caracteres en el archivo.	
void write(char[] cbuf, int off, int len)	Escribe una parte del array de caracteres, comenzando en la posición off y escribiendo len caracteres.	
void write(String str)	Escribe una cadena completa en el archivo.	
void write(String str, int off, int len)	Escribe una parte de la cadena, comenzando en la posición off y escribiendo len caracteres.	
void flush()	Fuerza a que se escriban en el archivo todos los caracteres que puedan estar en el buffer de salida.	
`FileWriter(String fileName)`	Si el archivo existe, se sobrescribe. Si no existe, se crea uno nuevo.	new FileWriter("C:/FicherosJava/texto.txt");
`FileWriter(String fileName, boolean append)`	Crea un flujo de salida FileWriter abriendo una conexión a un archivo cuyo nombre se pasa como cadena. Si append = true, se añaden los datos al final. Si append = false, se sobrescribe el contenido. Si no existe, se crea uno nuevo.	FileWriter fw = new FileWriter("C:/FicherosJava/texto.txt", true);
`FileWriter(FileDescriptor fd)`	Crea un flujo de salida FileWriter mediante un descriptor de fichero (FileDescriptor).	FileOutputStream fos = new FileOutputStream("texto.txt"); FileDescriptor fd = fos.getFD(); FileWriter fw = new FileWriter(fd);

FILEREADER

Constructor	Descripción	Ejemplo
`FileReader(File file)`	Crea un flujo de entrada FileReader abriendo una conexión a un archivo representado por un objeto File. <b Si el archivo no existe o no se puede abrir para lectura, se lanza una excepción.	java File f = new File("texto.txt"); FileReader fr = new FileReader(f);
`FileReader(FileDescriptor fd)`	Crea un flujo de entrada FileReader a partir de un descriptor de fichero (FileDescriptor).	java FileInputStream fis = new FileInputStream("texto.txt"); FileReader fr = new FileReader(fis.getFD());
`FileReader(String fileName)`	Crea un flujo de entrada FileReader abriendo una conexión a un archivo cuyo nombre se pasa como cadena. Si el archivo no existe o no se puede abrir, se lanza una excepción.	java FileReader fr = new FileReader("C:/texto.txt");

Modificador y método	Descripción
void close()	Cierra el flujo de entrada y libera todos los recursos del sistema asociados. Cualquier intento posterior de lectura generará una IOException.
int read()	Lee un único carácter del archivo y devuelve su código Unicode como entero. Devuelve -1 si se ha llegado al final del archivo.
int read(char[] cbuf)	Lee caracteres y los almacena en el array cbuf. Devuelve el número de caracteres leídos o -1 si se alcanzó el final del archivo.
int read(char[] cbuf, int off, int len)	Lee hasta len caracteres del archivo y los coloca en el array cbuf, comenzando en la posición off. Devuelve el número de caracteres leídos o -1 si no hay más datos.
boolean ready()	Indica si el flujo está listo para ser leído sin bloqueo. Devuelve true si hay caracteres disponibles, false en caso contrario.
long skip(long n)	Omite n caracteres en el flujo de entrada. Devuelve el número real de caracteres saltados.

4.3.1 Codificación de caracteres y errores habituales

En aplicaciones que intercambian o almacenan información entre distintos sistemas, es muy común encontrarse

con **problemas de codificación de caracteres**.

Estos problemas aparecen cuando el sistema que escribe y el que lee **no usan la misma norma de codificación**.

Codificación de caracteres.

La codificación de caracteres es el método que **permite convertir un caracter del lenguage natural**, el de los humanos, en **un símbolo de otro sistema de representación, aplicando una serie de normas** o reglas de codificación.

El ejemplo más gráfico suele ser el del código morse, cuyas reglas permiten convertir letras y números en señales (rayas y puntos) emitidas de forma intermitente.

En informática, las normas de codificación permiten que dos sistemas intercambien información usando el mismo código numérico para cada caracter.

Las normas más conocidas de codificación son las siguientes:

- **ASCII:** basado en el alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales. **Utiliza 7 bits** para representar los caracteres, aunque inicialmente empleaba un bit adicional (bit de paridad) que se usaba para detectar errores en la transmisión.

  Incluye, básicamente, letras mayúsculas y minúsculas del inglés, dígitos, signos de puntuación y caracteres de control, dejando fuera los caracteres específicos de los idiomas distintos del inglés, como por ejemplo, las vocales acentuadas o la letra ñ.

- **ISO-8859-1 (Latin-1):** es una **extensión del código ascii que utiliza 8 bits** para proporcionar caracteres adicionales usados en idiomas distintos al inglés, como el español.

  Existen 15 variantes y cada una cubre las necesidades de un alfabeto diferente: latino, europa del este, hebreo cirílico,...

  La norma **ISO-8859-15, es el Latin-1, con el caracter del euro**.

- **cp1252** (codepage 1252): Windows usa sus propias variantes de los estándares ISO.

  La cp1252 es compatible con ISO-8859-1, menos en los 32 primeros caracteres de control, que han usado para incluir, por ejemplo, el caracter del euro.

- **UTF-8:** es el formato de transformación **Unicode, de 8 bits de longitud variable**. Unicode es un estándar industrial cuyo objetivo es proporcionar el medio por el cual un texto en cualquier forma e idioma pueda ser codificado para el uso informático. Cubre la mayor parte de las escrituras usadas actualmente.

La misma norma de codificación que se use para escribir se debe utilizar para la lectura.

Esto tiene toda la lógica del mundo, puesto que si escribimos un fichero con ISO-8859-1 no debemos esperar que un sistema que lee en UTF-8 lo entienda sin más (aunque realmente entienda gran parte).

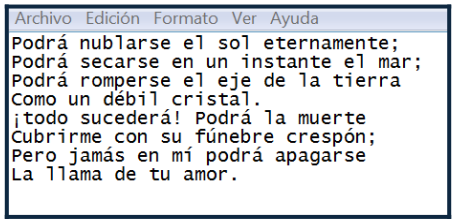
¿Por qué aparecen caracteres "raros"?:

Los caracteres "raros" aparecen por una conversión incorrecta entre dos codificaciones distintas. Se suelen producir porque se utiliza la codificación por defecto del sistema o programa y esta no coincide con la original o, directamente, por desconocimiento de la norma de codificación de la fuente de lectura.

Así, por ejemplo, podemos encontrarnos con los siguientes caracteres "raros" escribiendo la misma palabra:

- **España → EspaÃ±a: si escribimos en UTF-8 y leemos en ISO-8859-1**. La letra eñe se codifica en UTF-8 con dos bytes que en ISO-8859-1 representan la A mayúscula con tilde (Ã) y el símbolo más-menos (±).
- **España → Espa�a: si escribimos en ISO-8859-1 y leemos en UTF-8**. La codificación de la eñe en ISO-8859-1 es inválida en UTF-8 y se sustituye por un caracter de sustitución, que puede ser una interrogación, un espacio en blanco... depende de la implementación.

  Ejemplo: Tenemos un fichero de texto creado con el bloc de notas de Windows con codificación ASCII



Si intentamos leerlo con FileWiter, los acentos nos saldrán caracteres raros por que los métodos de esta clase leen caracteres Unicode.

Ejemplo:

```
public class EJCaracterAcentuadas {
    public static void main(String[] args) throws IOException {
       FileReader entrada=null;
       try{
           int cad;
            entrada= new FileReader("C:/FicherosJava/Copia.txt");
            while ((cad = entrada.read()) != -1)
            {      System.out.print((char)cad);        }
           }
        catch(FileNotFoundException e){
            System.out.println("error al abrir el fichero");    }
        catch(IOException e){
            System.out.printf("error de entrada/salida");      }
        finally{
            if (entrada!=null)
                    entrada.close();
            }
    }
}
```

Salida:

```
run:
Podr□ nublarse el sol eternamente;
Podr□ secarse en un instante el mar;
Podr□ romperse el eje de la tierra
Como un d□bil cristal.
□todo suceder□! Podr□ la muerte
Cubrirme con su f□nebre cresp□n;
Pero jam□s en m□ podr□ apagarse
La llama de tu amor.

BUILD SUCCESSFUL (total time: 0 seconds)
```

Para solucionarlo, podemos abrir el fichero para que se use, en este caso, la codificación **ISO-8859-15**

Las clases **InputStreamReader e OutputStreamReader**:

- Representan una conexión entre un stream de bytes y un stream de caracteres.
- Podemos leer bytes convertirlos a caracteres atendiendo a una codificación concreta (ISO Latin 1, UTF8,...).

```
public class EJCaracterAcentuadas {
    public static void main(String[] args) {
        try (InputStreamReader entrada = new InputStreamReader(
                new FileInputStream("C:/FicherosJava/Copia.txt"), "ISO-8859-1")) {

            int cad;
            while ((cad = entrada.read()) != -1) {
                System.out.print((char) cad);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
        } catch (IOException e) {
            System.out.println("Error de entrada/salida");
        }
    }}
```

4.4 Las clases OutputStreamReader e InputStreamReader.

En la librería **java.io**, además de las clases básicas de lectura/escritura de bytes (**InputStream, OutputStream**) y de caracteres (**Reader, Writer**), existen **clases puente que permiten convertir entre ambos mundos**:

- **InputStreamReader**: convierte un **flujo de bytes** (InputStream) en un **flujo de caracteres** (Reader).
- **OutputStreamWriter**: convierte un **flujo de caracteres** (Writer) en un **flujo de bytes** (OutputStream).

¿Por qué son necesarias?

- Los bytes son la forma más básica de entrada/salida en Java (adecuados para datos binarios: imágenes, audio, PDF…).
- Los caracteres son más adecuados para trabajar con texto, ya que Java internamente usa Unicode.
- Cuando queremos leer o escribir texto desde/hacia un flujo de bytes (por ejemplo, un archivo, un socket de red, etc.), necesitamos un puente que convierta esos bytes en caracteres y viceversa.
- Además, estas clases permiten especificar la codificación (UTF-8, ISO-8859-1, etc.), lo que evita los problemas de acentos y caracteres raros.

Constructor de InputStreamReader

Metodos de InputStreamReader

void	**close**() Cierra el fichero y libera sus recursos asociados.
String	**getEncoding**() Devuelve el nombre de la cdificación usada por el flujo.
int	**read**() Lee un caracter
int	**read**(char[] cbuf, int off, int len) Lee len caracteres de la corriente de entrada y los deposita en el vector de caracteres cbuf a paritr de off
boolean	**ready**() Indica si el flujo de entrada está listo para ser leído

Metodos de OutputStreamReader

void	**close**() cierra el flujo de salida y libera sus recursos asociados.
void	**flush**() Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
String	**getEncoding**() Devuelve el nombre de la cdificación usada por el flujo.
void	**write**(char[] cbuf, int off, int len) Escribe len caractees a partir del desplazamineto marcado por off.
void	**write**(int c) Escribe un caracter
void	**write**(String str, int off, int len) Escribe una porción de la cadena empezando en off y de longitud len.

Ej: de lectura de un fichero con codificación ASCII

```
public class EJCaracterAcentuadas {
    public static void main(String[] args) {
        try (InputStreamReader entrada = new InputStreamReader(
                new FileInputStream("C:/FicherosJava/Copia.txt"), "ISO-8859-1")) {

            int cad;
            while ((cad = entrada.read()) != -1) {
                System.out.print((char) cad);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
        } catch (IOException e) {
            System.out.println("Error de entrada/salida");
        }
    }}
```

Salida:

```
run:
ISO8859_15
Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
Como un débil cristal.
¡todo sucederá! Podrá la muerte
Cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
La llama de tu amor.

BUILD SUCCESSFUL (total time: 0 seconds)
```

4.5 Buffer para el flujo de caracteres: Clases BufferedReader y BufferedWriter

A la hora **de optimizar los flujos de caracteres**, existen las clases **BufferedReader** y **BufferedWriter** que crean un buffer **para agilizar las operaciones de lectura y escritura en flujos de caracteres**.

- La clase **BufferedReader** recibe un flujo de caracteres e implementa un buffer para poder leer líneas de texto.

  Define el método **readLine** para leer una línea de texto. Esta clase se utiliza si sabemos que el archivo es de texto y está escrito en líneas separadas por retornos de carro.

Constructor de BufferedReader
`BufferedReader(Reader in)` Crea un buffer para el flujo de caracteres de entrada utilizando el tamaño por defecto.
`BufferedReader(Reader in, int sz)` Crea un buffer para el flujo de caracteres de entrada utilizando el tamaño indicado en el parámetro sz. Lanza un IllegalArgumentException si sz es <= 0

Para el manejo de archivos hay dos formas para construir un objeto de tipo BufferedReader:

- Una es utilizar un objeto InputStreamReader creado sobre un objeto de tipo FileInputStream:

```
BufferedReader buffer = new BufferedReader (new InputStreamReader (
                                        new FileInputStream ("archivo.dat")));
```

- La otra forma es aceptar el tamaño del buffer y la codificación predefinidos, lo cuál muchas veces es lo más conveniente, y para estos casos se puede usar la clase FileReader que como argumento en el constructor se le pasa el nombre del archivo.

```
BufferedReader fd_in = new BufferedReader ( new FileReader ("archivo.dat"));
```

- La clase **BufferedWriter** escribe texto a un flujo de salida que acepte caracteres proporcionando un buffer para la escritura eficiente de caracteres, arreglos y strings.

  Define **el método write** para **escribir una línea de texto** y el método **newLine** para **escribir un salto de línea** de acuerdo al sistema operativo.

  Esta clase se utiliza si sabemos que el archivo es de texto y está escrito en líneas separadas por retornos de carro.

Constructor de BufferedWriter
`BufferedWriter(Writer out)` Crea un buffer para el flujo de caracteres de salida utilizando el tamaño por defecto.
`BufferedWriter(Writer out, int sz)` Crea un buffer para el flujo de caracteres de salida utilizando el tamaño indicado en el parámetro sz. Lanza un IllegalArgumentException si sz es <= 0

Para el manejo de archivos hay dos formas para construir un objeto de tipo BufferedWriter:

- Una es utilizar un objeto OutputStreamReader creado sobre un objeto de tipo FileOutputStream:

```
BufferedWriter fd_out = new BufferedWriter (new OutputStreamWriter
                                        (new FileOutputStream ("archivo.dat")));
```

La ventaja de hacerlo así es que se puede manejar el tamaño del buffer e incluso cambiar la codificación de los caracteres.

- La otra forma es aceptar el tamaño del buffer y la codificación predefinidos, lo cuál muchas veces es lo más conveniente, y para estos casos se puede usar la clase FileWriter que como argumento en el constructor se le pasa el nombre del archivo.

```
BufferedWriter fd_out = new BufferedWriter (new FileWriter ("archivo.dat"));
```

Ej de copiar un fichero de Texto ASCII en otro fichero utilizando buffers para mejorar las operaciones de lectura y escritura:

```
import java.io.*;

public class EjficheroTextoBuffer {
    public static void main(String[] args) {
        // try-with-resources con apertura en una sola instrucción
        try (
            BufferedReader bufferent = new BufferedReader(
                new InputStreamReader(
                    new FileInputStream("C:/FicherosJava/Copia.txt"), "ISO-8859-15"));

            BufferedWriter buffersal = new BufferedWriter(
                new OutputStreamWriter(
                    new FileOutputStream("C:/FicherosJava/Destino.txt"), "ISO-8859-15"))
        ) {
            String cad;
            while ((cad = bufferent.readLine()) != null) {
                System.out.println(cad);
                buffersal.write(cad);
                buffersal.newLine(); // mantiene los saltos de línea
            }
            buffersal.flush(); // asegura que todo se escriba en disco

        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
        } catch (IOException e) {
            System.out.println("Error de entrada/salida");
        }
    }
}
```

5. Acceso a datos primitivos:Clases DataInputStream y DataOuputStream

Aunque leer y escribir bytes es útil, a menudo es necesario transmitir datos de tipos primitivos dentro de un flujo. Las clases **DataInputStream** y **DataOutputStream** proporcionan **métodos para la lectura y escritura de tipos primitivos** (int, float, double, etc..) de un modo independiente de la máquina.

```
public class DataOutputStream
extends FilterOutputStream
implements DataOutput
```

```
public class DataInputStream
extends FilterInputStream
implements DataInput
```

Constructor de DataOutputStream	**Ejemplo:**
`DataOutputStream(OutputStream sal)` Se crea un objeto de la clase DataOutputStream vinculándolo a un objeto OutputStream para escribir en un archivo. Permite escribir un flujo de salida, datos de cualquier tipo primitivo (int, float, doublé, etc..)	`FileOutputStream fileSal=new FileOutputStream("pedido.txt");` `DataOutputStream Salida=new DataOuputStream(fileSal));`
Constructor de DataInputStream	**Ejemplo:**
`DataInputStream(InputStream in)` Se crea un objeto de la clase DataInputStream vinculándolo a un objeto InputStream para leer desde un archivo. Permite leer un flujo de entrada, datos de cualquier tipo primitivo. Solo es posible leer datos, escritos por DataOutputStream	`FileInputStream fileEnt=new FileInputStream("pedido.txt");` `DataInputStream entrada=new DataInputStream(fileEnt));`

A continuación se detallan los métodos de estas dos clases:

Modificador y tipo de DataOutputStream	Métodos y descripción
void	`flush()` throws IOException

	Vacia el buffer se salida. Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
int	**size**() Devuelve el número de bytes escritos en esta corriente de datos de salida hasta el momento.
void	**write**(byte[] b, int off, int len) throws IOException Escribe len bytes en la corriente de salida desde el array b **comenzando** en la posición off.
void	**write**(int b) throws IOException Escribe el primer byte de menor peso a corriente de salida. Los otros 3 restantes bytes se ignoran..
void	**writeBoolean**(boolean v) throws IOException Escribe un char en el flujo de salida ocupando 1 byte .
void	**writeByte**(int v) throws IOException Escribe un byte en el flujo de salida ocupando 1byte.
void	**writeBytes**(**String** s) throws IOException Escribe una cadena en el flujo de salida como una secuencia de bytes.
void	**writeChar**(int v) throws IOException Escribe un char en el flujo de salida ocupando 2 bytes.
void	**writeChars**(**String** s) throws IOException Escribe una cadena en el flujo de salida como una secuencia de caracteres. Comportamiento: para cada char de la cadena llama internamente a writeChar; escribe 2 bytes por caracter (UTF-16). Metadatos: ninguno de los dos escribe la longitud de la cadena; writeUTF sí lo hace (2 bytes de longitud y la codifica en modified UTF-8).
void	**writeDouble**(double v) throws IOException Escribe un valor numérico double en el flujo de salida ocupando 8bytes.
void	**writeFloat**(float v) throws IOException Escribe un valor numérico float en el flujo de salida ocupando 4bytes.
void	**writeInt**(int v) throws IOException Escribe un valor numérico entero en el flujo de salida ocupando 4bytes.
void	**writeLong**(long v) throws IOException Escribe un valor entero long en el flujo de salida ocupando 8bytes.
void	**writeShort**(int v) throws IOException Escribe un valor entero short en el flujo de salida ocupando 2bytes.
void	**writeUTF**(**String** str) throws IOException Escribe un cadena UNICODE utilizando codificación **UTF-8**. en el flujo de salida.

Modificador y tipo de DataInputStream	Métodos y descripción
int	**read**(byte[] b) throws IOException Lee b.length bytes del flujo de entrada y los guarda en el array pasado como parámetro. Devuelve el número total de bytes leidos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada.
int	**read**(byte[] b, int off, int len) throws IOException Lee len bytes del flujo de entrada y los guarda en el array a partir del desplazamiento off pasado como parámetro. Devuelve el número total de bytes leidos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada
boolean	**readBoolean**() throws IOException Lee un byte del flujo de entrada y devuelve verdadero si ese byte es distinto de cero, falso si ese byte es cero. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
byte	**readByte**() Lee un byte del flujo de entrada. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
char	**readChar**() throws IOException Lee dos bytes del flujo de entrada y devuelve un valor char. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
double	**readDouble**() throws IOException Lee ocho bytes del flujo de entrada y devuelve un valor numérico double. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
float	**readFloat**() throws IOException Lee cuatro bytes de entrada y devuelve un valor numérico float Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
void	**readFully**(byte[] b) throws IOException Lee b.length bytes del flujo de entrada y los deposita en el array b. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes. Es un método que fuerza la lectura de exactamente un número determinado de bytes desde un flujo de entrada hasta rellenar un array.
void	**readFully**(byte[] b, int off, int len) throws IOException Lee b.length bytes del flujo de entrada y los deposita en el array b a partir del desplazamiento off. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes.
int	**readInt**() throws IOException Lee cuatro bytes de entrada y devuelve un valor numérico int. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
String	**readLine**() throws IOException **Deprecated.** *Este método no convierte correctamente los bytes a caracteres. A partir de JDK 1.1, la mejor forma de leer las líneas de texto se realiza mediante el método BufferedReader.readLine (). Los programas que utilizan la clase DataInputStream para leer las líneas se pueden sustituir el código de la forma:* *DataInputStream d = new DataInputStream (en);* *Por:* *BufferedReader d = new BufferedReader (new InputStreamReader (en))*

long	**readLong**() throws IOException Lee ocho bytes de entrada y devuelve un valor numérico long Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
short	**readShort**() throws IOException Lee dos bytes de entrada y devuelve un valor numérico short. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
int	**readUnsignedByte**() throws IOException Lee un byte del flujo de entrada y devuelve un valor int en el rango de valores del 0 al 255. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
int	**readUnsignedShort**() throws IOException Lee dos byte del flujo de entrada y devuelve un valor int en el rango de valores del 0 al 65535. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
String	**readUTF**() throws IOException Lee una cadena Unicode en formato UTF-8 formato del flujo de entrada. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
static **String**	**readUTF**(**DataInput** in) throws IOException Lee una cadena Unicode en formato UTF-8 formato del flujo de entrada. Lanza una ***EOFException***, si el archivo llega al final antes de leer todos los bytes
int	**skipBytes**(int n) throws IOException Hace un intento de saltarse n bytes de datos del flujo de entrada, descarta los bytes omitidos. Sin embargo, puede saltar sobre un número menor de bytes, posiblemente cero. Este método no produce una EOFException.

5.1 Clases BufferedInputStream y BufferedOutputStream

Estas clases asignan un **buffer de memoria a los flujos de byte de I/O**.

- El buffer permite que Java **realice operaciones I/O por bloques** en lugar de byte a byte, reduciendo drásticamente las llamadas al sistema y mejorando el rendimiento al leer o escribir ficheros o sockets.

Qué hacen exactamente:

Agrupan **lecturas/escrituras pequeñas en operaciones más grandes** (hasta el tamaño del buffer).

- **Para lectura**: rellenan el buffer desde el dispositivo y sirven bytes desde memoria hasta agotarlo.
- **Para escritura**: acumulan bytes en memoria y los envían al dispositivo en bloque cuando el buffer está lleno o cuando se hace flush/close.

Se usan **envolviendo cualquier InputStream/OutputStream**:

```
new BufferedInputStream(in) / new BufferedOutputStream(out).
```

Constructor de BufferedInputStream
BufferedInputStream(**InputStream** in) Crea un objeto BufferedInputStream con el tamaño de buffer por defecto
BufferedInputStream(**InputStream** in, int size) Crea un objeto BufferedInputStream con el tamaño de buffer especificado en el argumento size. Lanza una IllegalArgumentException si size <= 0
Constructor de BufferedOutputStream
BufferedOutputStream(**OutputStream** out) Crea un objeto BufferedOutputStream con el tamaño de buffer por defecto
BufferedOutputStream(**OutputStream** out, int size) Crea un objeto BufferedOutputStream con el tamaño de buffer especificado en el argumento size. Lanza una IllegalArgumentException si size <= 0.

Metodos de BufferedInputStream	
int	**available**() throws IOException Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	**close**() throws IOException Cierra la corriente de entrada y libera los recursos del sistema que esté usando
void	**mark**(int readlimit) Marca la posición actual de la corriente de entrada.
boolean	**markSupported**() Prueba si esta corriente de entrada soporta los métodos mark y reset.
int	**read**() throws IOException

	Lee el siguiente byte de la corriente de entrada.
int	**read**(byte[] b, int off, int len) throws IOException Lee b.length bytes del flujo de entrada y los guarda en el array pasado como parámetro. Devuelve el número total de bytes leidos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada.
void	**reset**() throws IOException Coloca la corriente de entrada en la posición que tenía la última vez que se invocó `mark`.
long	**skip**(long n) throws IOException Salta y descarta los siguientes n bytes de la corriente de entrada.

Métodos heredados de la classe java.io.FilterInputStream:read

Metodos de BufferedOutputStream	
void	**flush**() throws IOException Vacia el buffer se salida. Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
void	**write**(byte[] b, int off, int len) throws IOException Escribe `len` bytes en la corriente de salida desde el array b comenzando en la posición off.
void	**write**(int b) throws IOException Escribe el primer byte de menor peso a corriente de salida. Los otros 3 restantes bytes se ignoran..

A continuación veremos un ejemplo que combinan las clases que tratan directamente sobre los flujos de entrada o de salida con las clases que las envuelven.

Ej: leer un archivo de números reales con buffer

```
public class EjFicheroDatosRealesConBuffer {
    public static void main(String[] args) {
        Path fichero = Path.of("C:/FicherosJava/FicheroDoubleBuffer.bin");
        Random rnd = new Random(); // Crear una única instancia de Random (no dentro del
                                   //bucle)

        // Escritura: DataOutputStream envuelto en BufferedOutputStream para rendimiento
        try (DataOutputStream out = new DataOutputStream(
                new BufferedOutputStream(new FileOutputStream(fichero.toFile())))) {

            for (int i = 0; i < 100; i++) {
                double valor = rnd.nextDouble();
                out.writeDouble(valor); // Escribe 8 bytes
            }

            // try-with-resources cierra y hace flush automáticamente al salir del bloque
        } catch (FileNotFoundException e) {
           System.err.println("No se encontró el archivo para escritura: " + e.getMessage());
            return;
        } catch (IOException e) {
            System.err.println("Error al escribir el fichero: " + e.getMessage());
            return;
        }
        System.out.println("Lectura del archivo binario de números reales con buffer:");
        // Lectura: DataInputStream envuelto en BufferedInputStream
        DecimalFormat df = new DecimalFormat("#0.00");
        try (DataInputStream in = new DataInputStream(
                new BufferedInputStream(new FileInputStream(fichero.toFile())))) {
                        while (true) {
                try {
                    double d = in.readDouble(); // Lee 8 bytes y devuelve un double; lanza
                                                //EOFException al final
                  System.out.printf("%6s%s", df.format(d),
                } catch (EOFException eof) {
                   // EOFException indica fin de fichero para lecturas secuenciales binarias
                    break;
                }
            }
            System.out.println(); // Salto de línea final tras la impresión de datos
        } catch (FileNotFoundException e) {
            System.err.println("No se encontró el archivo para lectura: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error al leer el fichero: " + e.getMessage());
        }
    }}
```

6. Serialización

La serialización de un objeto consiste en obtener una secuencia de bytes que represente el estado de dicho objeto. Esta secuencia puede utilizarse de varias maneras (puede enviarse a través de la red, guardarse en un fichero para su uso posterior, utilizarse para recomponer el objeto original, etc.).

Serialización: *Posibilidad de escribir/leer Objetos java en Streams.*

6.1 Interfaz Serializable

Un **objeto serializable** es un objeto que se puede **convertir en una secuencia de bytes** para poder ser tratado por un stream.

Para que un objeto **sea serializable**, debe implementar la interfaz **java.io.Serializable**.

```
import java.io.Serializable;
public class Datos implements Serializable
{  private int a;
   private String b;
   private char c;
}
```

- **Esta interfaz no define ningún método**. Simplemente se usa para 'marcar' aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruídas). se trata de un interface vacío. No tiene métodos que debamos redefinir.

```
import java.io.*;
public interface Serializable{
}
```

- Objetos tan comunes como String, Vector o ArrayList implementan Serializable, de modo que pueden ser serializados y reconstruidos más tarde.
- El sistema de ejecución de Java **se encarga de hacer la serialización de forma automática**.
- **Solo los objetos que implemente la interfaz Serializable pueden ser escritos a stream**.

  La clase de cada objeto es codificada incluyendo el nombre de la clase y la firma (su prototipo), los valores de sus campos, y cualquier objeto referenciado desde el objeto inicial, o sea, se guarda también una cabecera junto al objeto en el fichero.
- Por razones de seguridad, las clases no son serializables por defecto.

6.2 Excluir campos al serializar objetos

Algunas veces es necesario **excluir campos a la hora de serializar objetos**, por ejemplo cuando se tiene un objeto que guarda la información de un usuario incluida su contraseña.

Para **evitar que esos campos** sean serializados basta con utilizar el modificador **transient**.

```
private  String nombre;
private transient String contrasenia; //Este campo no será guardado
```

6.3 Campos estáticos (static) y serialización

Qué es un campo estático

- Un **campo estático** pertenece a la **clase**, no a sus instancias; hay **una sola copia compartida por todas las**

instancias.

- Se declara con la palabra clave **static**, p. ej. private static int contador;.

¿Se serializan los campos estáticos?

- **No.** La serialización con ObjectOutputStream guarda el **estado de las instancias** (campos de instancia no transient), **pero no incluye campos static**.
- Motivo: los campos estáticos no forman parte del estado propio del objeto, sino del estado de la clase cargada en la JVM lectora.

Consecuencias prácticas

- Si escribes objetos a disco y esperas que un campo static tenga el mismo valor al deserializar en otra ejecución, no ocurrirá automáticamente. Tras la deserialización el valor static será el que tenga la clase en la JVM lectora (valor por defecto o modificado por inicializadores estáticos o código de arranque).
- **Cambios en campos static durante la ejecución no se reflejan en el stream ni se restauran al deserializar.**

```
public class Producto {
    public static int contador = 0;           // contador compartido. No se serializa
    public static final double IVA = 0.21;    // constante. No se seraliza
    private String nombre;
```

6.4 Qué se guarda exactamente al serializar un objeto

Cuando Java **serializa un objeto con ObjectOutputStream** no guarda solo los valores de sus campos: **escribe también metadatos** necesarios para que el receptor pueda reconstruir el objeto correctamente.

Esos elementos son:

- **Descriptor de la clase**
  - **Nombre completo de la clase** (package + clase).
  - Información sobre la **firma de la clase**: campos (nombre, tipo), modificadores relevantes (private, public, etcc) y la versión de la clase (**serialVersionUID**).
  - Esta información se **empaqueta en un objeto interno (ObjectStreamClass) y se escribe como parte del stream.**
- **serialVersionUID**
  - **Valor numérico que identifica la versión binaria de la clase.**
  - Puede ser cualquiera que sea **un long** válido.
  - Se escribe con el descriptor y se usa **en la deserialización para comprobar compatibilidad entre la clase escrita y la clase disponible en la JVM lectora**.

  Funcionamiento
  - *Cuando se serializa un objeto de clase Serializable, Java Runtime asocia un número de versión en serie (llamado serialVersionUID) con este objeto serializado.*
  - *En el momento en que deserializa este objeto serializado, Java Runtime comprueba con el serialVersionUID del objeto serializado con el serialVersionUID de la clase.*
  - *Si ambos son iguales, entonces solo procede con el proceso adicional de deserialización, sino que lanza InvalidClassException.*
  - *Con este código numérico asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales.*

```
Public class Persona implements Serializable {
    private static final long serialVersionUID = 1L;
                                    // <-- declarar siempre para control de versiones
    private String nombre;
    private LocalDate nacimiento;
    private transient String contrasenia; // no se serializa
}
```

Estado de los campos

- **Valores de los campos no estáticos y no transient** en el orden y tipo definidos por la clase.

Grafo de objetos referenciados

- **Cualquier objeto referenciado transitivamente** (campos que apuntan a otros objetos) también se serializa automáticamente y se incluye en el stream.
- Para evitar duplicar bytes, el mecanismo mantiene una **tabla de referencias**: si el mismo objeto aparece varias veces se escribe una única vez y las siguientes referencias se representan por punteros internos.

Cabeceras de stream y descriptores auxiliares

- Al **comienzo del stream ObjectOutputStream** escribe un **encabezado general del stream** (stream header).
- Además, cuando se escriben clases nuevas durante el stream, se emiten descriptores de clase adicionales para identificar cambios.

Problema con las cabeceras al añadir objetos a un fichero serializado

- **ObjectOutputStream** escribe al crearse un encabezado (**stream header**).
- **Si cierras un OOS** y luego **abres otro sobre el mismo fichero en modo append**, el **segundo OOS vuelve a escribir su cabecera**.
- Al leer con **ObjectInputStream esa cabecera adicional se interpreta como datos inesperados** y provoca errores como **StreamCorruptedException** o fallos al readObject.

Solución: Sobrescribe la escritura de la cabecera para que no escriba nada.

```
/**
 * ObjectOutputStreamNoHeader: evita escribir el stream header al abrirse.
 * Útil para abrir un fichero en modo append y añadir objetos sin reescribir la cabecera.
 */
public class ObjectOutputStreamNoHeader extends ObjectOutputStream {
    /** Constructor principal que envuelve un OutputStream */
    public ObjectOutputStreamNoHeader(OutputStream out) throws IOException {
        super(out);
    }
    /** Constructor protegido sin argumentos (delegado a la superclase) */
    protected ObjectOutputStreamNoHeader() throws IOException, SecurityException {
        super();
    }
    /** Sobrescribe la escritura de la cabecera para que no haga nada. */
    @Override
    protected void writeStreamHeader() throws IOException {
        // No escribir header al abrir en modo append
            }
}
```

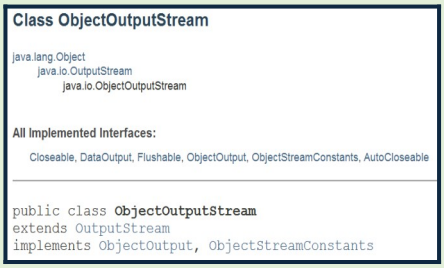
Ejemplos: Tenemos objetos de la clase persona descrita anteriormente.

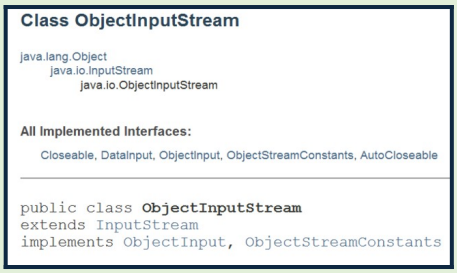
```
// 1) Primera escritura: crear fichero y escribir objetos con OOS normal (escribe cabecera)
private static final File fichero= new File("Personas.dat");

List<Persona> inicio = List.of(
       new Persona("Ana", LocalDate.of(1990, 1, 5), "pwAna"),
       new Persona("Luis", LocalDate.of(1985, 6, 20), "pwLuis")
        );

try (ObjectOutputStream oos = new ObjectOutputStream(
      new BufferedOutputStream(new FileOutputStream(fichero)))) {
       // Escribimos una colección (ArrayList) o los objetos individualmente
            oos.writeObject(inicio); // escribe cabecera + descriptor + datos
            System.out.println("Primera escritura completada.");
        } catch (IOException e) {
            e.printStackTrace();
```

```
            return;
        }

    // 2) Append posterior: abrir en modo append y usar ObjectOutputStreamNoHeader
        List<Persona> anhadidos = List.of(
                new Persona("María", LocalDate.of(1992, 3, 15), "pwMaria"),
                new Persona("Carlos", LocalDate.of(1978, 12, 2), "pwCarlos")
        );

        try (FileOutputStream fos = new FileOutputStream(fichero, true);
             BufferedOutputStream bos = new BufferedOutputStream(fos);
             ObjectOutputStreamNoHeader oos2 = new ObjectOutputStreamNoHeader(bos)) {

            // Al usar ObjectOutputStreamNoHeader no se reescribe la cabecera inicial
            oos2.writeObject(anhadidos);
            System.out.println("Append completado.");
        } catch (IOException e) {
        }
```

6.5 Flujos para la entrada y salida de objetos: ObjectInputStream e ObjectOutputStream

- La **serialización** de objetos en Java es el **proceso de convertir un objeto en una secuencia de bytes** para que pueda ser almacenado en un archivo. Este proceso es esencial para la persistencia de objetos.





- La serialización está basada en la jerarquía de clases de **InputStream y OutputStream**, que se ocupan de los flujos de datos en Java. Estas clases **están orientadas a bytes**, lo que significa que trabajan con **datos binarios**.
- **ObjectInputStream y ObjectOutputStream** son clases que implementan las interfaces ObjectInput y ObjectOutput, respectivamente.

  DataInput/Output: Se usan para leer y escribir tipos de datos primitivos (por ejemplo, int, float, char, etc.), mientras que **ObjectInput/Output son específicamente para objetos**.

Características de los flujos de objetos:

- **Mezcla de valores primitivos y objetos**:

  Los flujos de objetos pueden contener no solo objetos serializados, sino también valores primitivos (como int, boolean, etc.) que son leídos o escritos con los métodos de DataInput y DataOutput.

- **Compatibilidad con la serialización**:

  Para que un objeto sea serializado, la clase del objeto debe implementar la interfaz Serializable. Si no implementa esta interfaz, se lanzará una NotSerializableException.

- **Proceso de serialización**:

  Un objeto **se convierte en un flujo de bytes utilizando ObjectOutputStream**.

  Posteriormente, este flujo puede ser transmitido, guardado o almacenado.

- **Para deserializar el objeto,**

Se utiliza ObjectInputStream, que reconstruye el objeto original a partir del flujo de bytes.

Serialización de objetos anidados:

- Si un objeto contiene otros objetos, estos **también deben ser serializables** para que puedan ser correctamente serializados y deserializados.
- La serialización está orientada a bytes, por lo tanto, se utilizan clases que están en la jerarquía de inputStream y OutputStream.

Para serializar un objeto:

Es necesario crear algún objeto de tipo ***OutputStream*** que se le pasa al constructor de **ObjectOutputStream**. A continuación se puede llamar algún metódo, por ejemplo, writeObject(), para serializar el objeto.

Constructor	Descripición
protected	`ObjectOutputStream(OutputStream out)` throws IOException Crea un ObjectOutputStream que escribe en el OutputStream especificado.

Ejemplo: Para escribir un objeto en un fichero:

```
FileOutputStream fs=new FileOutputStream(fichero);
ObjectOutputStream os = new ObjectOutputStream(fs));

O de forma abreviada:
ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream(fichero));
```

- **Para recuperar un objeto:**

Es necesario crear algún objeto de tipo ***InputStream*** que se le pasa al constructor de **ObjectInputStream**. A continuación se puede llamar algún metódo, por ejemplo, readObject(), para leer el objeto.

Constructor	Descripición
protected	`ObjectInputStream(InputStream en)` throws IOException Recupera datos primitivos y objetos previamente almanenados con ObjectOutputStream

```
FileInputStream fe=new FileInputStream(fichero);
ObjectInputStream oe = new ObjectInputStream(fe));

O de forma abreviada:
ObjectInputStream oe = new ObjectInputStream(new FileInputStream(fichero));
```

Hay que tener claro el orden y el tipo de los objetos almacenados en el disco para recuperarlos en el mismo orden.

6.5.1 Escritura de objetos en ficheros

La clase **ObjectOutputStream** permite crear objetos que se asocian a un objeto **FileOutputStream** y facilita métodos para escribir o almacenar secuencialmente información codificada en binario en el fichero asociado a dicho objeto:

Algunos métodos son:

Modifier and Type	Method and Description
void	`close()` Cierra el flujo de salida.
void	`defaultWriteObject()` Escribe los campos no estáticos y no transient de la clase actual en el flujo de salisa.
protected void	`drain()` Vacia todos los datos almacenados en el buffer de ObjectOutputStream.
protected boolean	`enableReplaceObject(boolean enable)` Activa el flujo de salida para la sustitución de objetos en el flujo.

void	flush() Vacía el flujo de salida
protected Object	replaceObject(Object obj) Este método permitirá a las subclases de ObjectOutputStream sustituir un objeto por otro durante la serialización.
void	reset() Reset will disregard the state of any objects already written to the stream.
void	write(byte[] buf) Escribe un array de bytes en el flujo de sallida
void	write(byte[] buf, int off, int len) Escribe un array de bytes en el flujo de sallida a partir del desplazamiento off
void	write(int val) Escribe un byte.
void	writeBoolean(boolean val) Escribe boolean.
void	writeByte(int val) Escribe 1 byte.
void	writeBytes(String str) Escribe un String como secuencia de bytes
void	writeChar(int val) Escribe un char con 16 bit.
void	writeChars(String str) Escribe un String como secuencia de caracteres
protected void	writeClassDescriptor(ObjectStreamClass desc) Escribe el descriptor de la clase especificada en el ObjectOutputStream
void	writeDouble(double val) Writes un double (64 bit)
void	writeFields() Escribe los campos del buffer en el flujo de salida.
void	writeFloat(float val) Escribe un float (32 bit).
void	writeInt(int val) Escribe un int (32 bit).
void	writeLong(long val) Escribe un long (64 bit).
void	**writeObject(Object obj)** **Escribe el objeto especificado en el ObjectOutputStream.**
protected void	writeObjectOverride(Object obj) Metodo usuado por las subclases para sobreescribir el método defaultwriteObject.
void	writeShort(int val) Escribe un short (16 bit).
void	writeUTF(String str) Escribe un string con el format UTF-8

Ejemplo: Escritura de objetos punto en un fichero.

```
package ejserializacionobjetosfichero;
import java.io.*;
class punto implements Serializable
{ private static final long serialVersionUID = 1L
 int x,y;
punto(int x,int y){
    this.x=x;     this.y=y;    }
public int getX(){    return x;}
public int getY(){    return y;}
}//fin clase punto

public class EjSerializacionObjetosFichero {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
     ObjectOutputStream salida=
 new ObjectOutputStream (new FileOutputStream("C:/FicherosJava/ejobjetos.dat"));

   salida.writeObject(new punto(5,8));
   salida.writeObject(new punto(1,9));
   salida.writeObject(new punto(3,7));
   salida.close();
   System.out.println("fin de la grabación de objetos");
   }
}
```

6.5.2 Lectura de objetos en ficheros

La clase **ObjectInputStream** permite crear objetos que se asocian a un objeto **FileInputStream** y facilita métodos para leer de él secuencialmente información codificada en binario:

Algunos métodos son:

Modificador y tipo	Metodo y descripición
`int`	**`available`**`()` Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
`void`	**`close`**`()` Cierra el flujo de entrada
`void`	**`defaultReadObject`**`()` Lee los campos no estáticos y no transient de la clase actual en el flujo de entrada
`protected boolean`	**`enableResolveObject`**`(boolean enable)` Activa el flujo de entra para que los objetos leídos puedan ser sustituidos.
`int`	**`read`**`()` Lee un byte de datos
`int`	**`read`**`(byte[] buf, int off, int len)` Reads into an array of bytes.
`boolean`	**`readBoolean`**`()` Lee un boolean.
`byte`	**`readByte`**`()` Lee un byte (8 bits).
`char`	**`readChar`**`()` Lee un carácter con 16bits.
`protected` **`ObjectS-treamClass`**	**`readClassDescriptor`**`()` Lee el descriptor desde el flujo de serialización.
`double`	**`readDouble`**`()` Lee un double (64 bit).
`float`	**`readFloat`**`()` Lee un float (32 bit).
`void`	**`readFully`**`(byte[] buf)` Lee buf.length bytes y los deposita en el array buf.
`void`	**`readFully`**`(byte[] buf, int off, int len)` Lee buf.length bytes y los deposita en el array buf a partir del desplazamiento off.
`int`	**`readInt`**`()` Lee un int (32 bit).
`String`	**`readLine`**`()` **Deprecated.**
`long`	**`readLong`**`()` Lee un long (64 bit).
`Object`	`readObject()` Lee un objeto desde el flujo ObjectInputStream.
`short`	**`readShort`**`()` Lee un short (16 bit).
`int`	**`readUnsignedByte`**`()` Lee un byte sin signo (8 bit)
`int`	**`readUnsignedShort`**`()` Lee un short sin signo (16 bit).
`String`	**`readUTF`**`()` Lee un String en formato **UTF-8**
`int`	**`skipBytes`**`(int len)` Salta len bytes.

Ej: Lectura de los objetos punto escrito en el anterior fichero.

```
ObjectInputStream  entrada=new ObjectInputStream
                        (new FileInputStream("C:/FicherosJava/ejobjetos.dat"));
   try
   {
       while(true){
         punto puntoentrada=(punto)entrada.readObject();
         System.out.println("(x="+puntoentrada.getX()+",
                             y="+puntoentrada.getY()+")");       }
   }
//cuando se llega al final se lanza una excepción
   catch (IOException e){
        System.out.println("llegado al final"); }
   entrada.close();
```

Salida:

```
run:
leyendo el fichero
(x=5, y=8)
(x=1, y=9)
(x=3, y=7)
llegado al final
BUILD SUCCESSFUL (total time: 1 second)
```

6.5.3 Serialización de objetos compuestos

Si dentro de la **clase hay atributos que son otras clases**, éstas clases a su vez también ***deben ser Serializable***.

Con los tipos de java (String, Integer, etc.) no hay problema porque lo son. Las principales clases en java ya son serializables.

Si ponemos como **atributos nuestras propias clases, éstas a su vez deben implementar Serializable**. Si no java lanza una excepción, por ejejmplo: Exception in thread "main" **java.io.NotSerializableException**: ejserializacionficherosobjetoscompuestos.Punto

Ejemplo: la clase Rectangulo tiene un atributo de tipo Punto. La clase Punto debe implementar también la interfaz serializable:

```
package ejserializacionficherosobjetoscompuestos;
import java.io.*;
class Punto implements Serializable
{   private static final long serialVersionUID = 1L
   int x,y;
    Punto(int x,int y){
        this.x=x;
        this.y=y;    }
    public int getX(){ return x;}
    public int getY(){ return y;}
}
class Rectangulo  implements java.io.Serializable{
    private static final long serialVersionUID = 1L
    private int ancho ;
    private int alto ;
    private Punto origen;
    public Rectangulo(int x, int y, int ancho, int alto){
        origen=new Punto(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public int getX(){return origen.getX();
    }
    public int getY(){return origen.getY();}
    public int getAncho(){return ancho;}
    public int getAlto(){return alto;   }
}
public class EjSerializacionFicherosObjetosCompuestos {
    public static void main(String[] args)
                    throws IOException, ClassNotFoundException {
      ObjectOutputStream salida=new ObjectOutputStream
                      (new FileOutputStream("C:/FicherosJava/ejobjetos2.dat"));
      salida.writeObject(new Rectangulo(3,4,5,8));
      salida.writeObject(new Rectangulo(6,1,5,8));
      salida.writeObject(new Rectangulo(2,4,5,8));
      salida.close();
      System.out.println("leyendo el fichero");
      ObjectInputStream  entrada=new ObjectInputStream
                      (new FileInputStream("C:/FicherosJava/ejobjetos2.dat"));
      try
       { while(true){
           Rectangulo Rectentrada=(Rectangulo)entrada.readObject();
           System.out.println("(x="+Rectentrada.getX()+",
                              y="+Rectentrada.getY()+"), alto="+
                     Rectentrada.getAlto()+" Ancho="+Rectentrada.getAncho());
       }
       }//Si llega al final se produce una excepición IOException
       catch (IOException e){
        System.out.println("llegado al final");        }
```

```
            entrada.close();
        }}
```

Salida:

```
run:
leyendo el fichero
(x=3, y=4), alto=8 Ancho=5
(x=6, y=1), alto=8 Ancho=5
(x=2, y=4), alto=8 Ancho=5
llegado al final
BUILD SUCCESSFUL (total time: 0 seconds)
```

6.5.4 Serialización de objetos con herencia

En Java, cuando una **subclase hereda de una superclase**, los **atributos de la superclase también son serializados** si la clase base (superclase) es **Serializable**. Por lo tanto, en el caso de la herencia, tanto los atributos de la subclase como los de la superclase se serializan.

- **No es necesario volver a implementar la interfaz Serializable** en la subclase (Perro) si ya está implementada en la superclase (Animal). Java permite que la **serialización se herede** de la superclase, lo que significa que si la superclase implementa Serializable, la subclase lo hereda automáticamente.
- serialVersionUID: Es una buena práctica definir serialVersionUID en la superclase y en la subclase para garantizar la compatibilidad entre versiones. No es estrictamente necesario en la subclase, pero se recomienda para evitar problemas de deserialización.

```
// Clase base (superclase) que implementa Serializable
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;  // Asegura la compatibilidad de versiones
     String nombre;
     // Constructor de la clase Animal
     public Animal(String nombre) {
         this.nombre = nombre;
     }
     public void hacerSonido() {
         System.out.println("El animal hace un sonido");
     }
}

// Clase derivada (subclase) que hereda de Animal
class Perro extends Animal {
    private static final long serialVersionUID = 1L;  // Asegura la compatibilidad de versiones
     String raza;
     // Constructor de la clase Perro
     public Perro(String nombre, String raza) {
         super(nombre);  // Llamada al constructor de la superclase
         this.raza = raza;
     }
     @Override
     public void hacerSonido() {
         System.out.println("El perro hace guau");
     }
     public void mostrarRaza() {
         System.out.println("Raza del perro: " + raza);
     }
}
```

Grabar un objeto perro

```
// Crear un objeto de la clase Perro
        Perro perro = new Perro("Max", "Labrador");

        try {
            // Crear un flujo de salida para escribir el objeto en un archivo binario
            ObjectOutputStream salida = new ObjectOutputStream(
                    new FileOutputStream("perros.dat"));
```

```
            // Serializar los objetos
            salida.writeObject(perro);
            salida.close();
            System.out.println("Objeto serializado correctamente.");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```