TEMA: FICHEROS

Contenido

1. Conceptos Básicos de Ficheros

Hasta ahora todos los datos que creábamos en nuestros programas solamente existían durante la ejecución de los mismos. Cuando salíamos del programa, todo lo que habíamos generado se perdía.

A veces nos interesa que la vida de los datos vaya más allá de la de los programas que los generaron. Es decir, que al salir de un programa los datos generados queden guardados en algún lugar que permita su recuperación desde el mismo u otros programas. Esto que quiere decir que queremos que los datos sean ***persistentes***.

En este tema veremos el uso básico de **archivos/ficheros** 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.
Además, el uso de esas bibliotecas nos obligará a introducir algunos conceptos "avanzados" de programación en Java para transformar nuestros datos a vectores de bytes.

Cuando se desea guardar información más allá del tiempo de ejecución de un programa, lo habitual es organizar esa información en uno o varios **ficheros** almacenados en algún soporte de almacenamiento persistente.

Otras posibilidades como el uso de **bases de datos** también utilizan archivos como soporte para el almacenamiento de la información.

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 0´s y 1´s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD...
Esa versión de bajo nivel, si bien es completamente cierta, desde el punto de vista de la programación de aplicaciones, es demasiado simple. Por ello definiremos varios criterios para distinguir diversas subcategorías de archivos.

Estos diferentes tipos de archivos se diferenciarán desde el punto de vista de la programación: Cada uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación.

1.1.- Paquete java.io

OBJETO FILE

En el paquete java.io se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones **a nivel de sistema** de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre...).

Aclaración: "No proporciona métodos de acceso a los archivos" quiere decir que no sirven para crear un fichero y poder acceder a él para escribir o leer su contenido, sino que sirven para ver información del fichero desde el punto de vista del sistema operativo.

Un **objeto File** puede representar un archivo o un directorio y sirve para obtener información (permisos, tamaño, …) y también para navegar por la estructura de archivos.

Su ***constructor*** puede recibir un único parámetro: una cadena que representa una ruta (path) en el sistema de archivos. O también puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

Si el archivo o carpeta que se intenta examinar no existe, la clase File **no devuelve** una excepción. Por lo que, si necesitamos saberlo, hay que utilizar el método **exists**. Este método devuelve true si la carpeta o archivo existe. Vamos a ver un ejemplo.

> Ojo, que **new File()** no crea el fichero físicamente, ¡es un mero descriptor que apunta a la información del fichero al que apunta o apunte en el futuro!

```java
public class InfoFichero0 {

    public static void main(String args[]) {
        if (args.length == 1) {
            File f = new File(args[0]);
            System.out.println("Nombre: " + f.getName());
            System.out.println("Camino: " + f.getPath());
            if (f.exists()) {
                System.out.print("Fichero existente ");
                System.out.print((f.canRead() ? " y se puede Leer" : " y no se puede leer"));
                System.out.print((f.canWrite() ? " y se puede Escribir" : " y no se puede escribir"));
                System.out.println(".");
                System.out.println("La longitud del fichero es de " + f.length() + " bytes");
            } else
                System.out.println("El fichero no existe.");

        } else
            System.out.println("Debe indicar un fichero.");
    }
}
```

Según la ayuda del api, los constructores son:

File (File padre, String hijo)	Crea una instancia de File a partir de un File padre y una ruta que se une a la ruta del hijo (String)
File (String ruta)	Crea una instancia de File usando la cadena recibida como ruta
File (String padre, String hijo)	Crea una instancia de File usando las 2 cadenas recibidas y a ruta es la concatenación de la primera con la segunda
File (URI uri)	Crea una instancia de File convirtiendo la URI en una ruta

```java
public class EjemploFile {

    public static void main(String[] args) {

        try {
            File f1 = new File("C:\\Pruebas");
            System.out.println(f1.getName());

            File f2 = new File("C:\\Pruebas\\prueba.txt");
            System.out.println(f2.getName());

            File dir = new File("C:\\Pruebas");
            File f3 = new File(dir, "prueba.txt");
            System.out.println(f3.getName());

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Arriba tienes ejemplo de los 3 primeros constructores, pruébalo tecleando el programa en tu Eclipse. Un ejemplo de la última, que quizás sea la que puedes ver menos clara, sería:

```
package IO.infoFile;
```

```
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;

public class EjemploURI {

	public static void main(String[] args) {
		File aFile;
		try {
			//aFile = new File(new URI("file:///c:/a.bat"));
			aFile = new File("https://elpais.com");
			System.out.println(aFile.getName());// false
		} catch (Exception e) {
			e.printStackTrace();
		}

	}
}
```

Creación de directorios/carpetas.

Esta necesita que exista la carpeta contenedora:

boolean java.io.File.mkdir()

Creates the directory named by this abstract pathname.

Returns:
`true` if and only if the directory was created; `false` otherwise
Throws:
SecurityException - If a security manager exists and its `java.lang.SecurityManager.checkWrite (java.lang.String)` method does not permit the named directory to be created

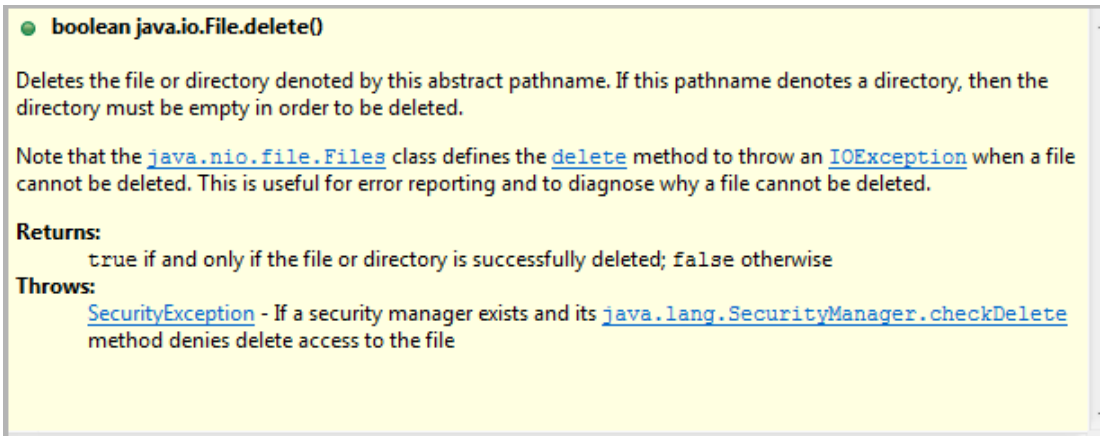
Esta crea la carpeta contenedora si no existe:

boolean java.io.File.mkdirs()

Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

Returns:
`true` if and only if the directory was created, along with all necessary parent directories; `false` otherwise
Throws:
SecurityException - If a security manager exists and its `java.lang.SecurityManager.checkRead (java.lang.String)` method does not permit verification of the existence of the named directory and all necessary parent directories; or if the `java.lang.SecurityManager.checkWrite (java.lang.String)` method does not permit the named directory and all necessary parent directories to be created

Borrado de ficheros/directorios:



boolean java.io.File.delete()

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.

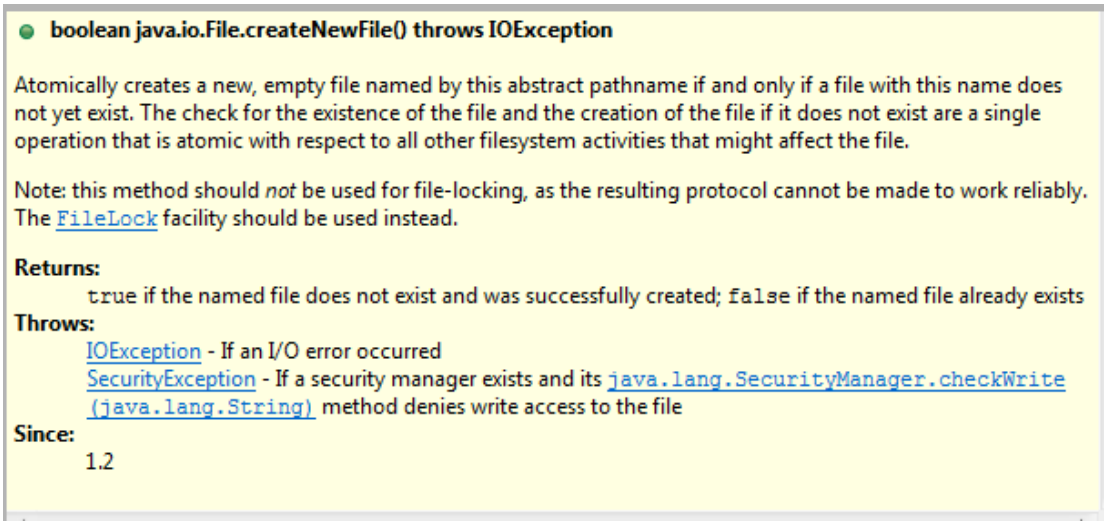
Note that the `java.nio.file.Files` class defines the `delete` method to throw an `IOException` when a file cannot be deleted. This is useful for error reporting and to diagnose why a file cannot be deleted.

Returns:
`true` if and only if the file or directory is successfully deleted; `false` otherwise

Throws:
SecurityException - If a security manager exists and its `java.lang.SecurityManager.checkDelete` method denies delete access to the file

Se borra el fichero si la ruta es de un fichero (deja la carpeta contenedora), o la carpeta si se refiere a la ruta de una carpeta. No borra el resto de carpetas: ¡Ojo, que solo se borra la carpeta si está vacía! Haz la prueba.

Creación de ficheros, crea el fichero si no existe.



boolean java.io.File.createNewFile() throws IOException

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The `FileLock` facility should be used instead.

Returns:
`true` if the named file does not exist and was successfully created; `false` if the named file already exists

Throws:
IOException - If an I/O error occurred
SecurityException - If a security manager exists and its `java.lang.SecurityManager.checkWrite (java.lang.String)` method denies write access to the file

Since:
1.2

Ejemplo de creación de un fichero:

```
import java.io.*;

public class CreaFichero1 {
	public static void main(String args[]){
		// Crea un objeto File dada la ruta completa
		File f1 = new File("C:\\Ficheros\\nuevo.txt");
		try {
		// A partir del objeto File creamos el fichero físicamente
		 if (f1.createNewFile())
			System.out.println("El fichero se ha creado correctamente");
		 else
			System.out.println("No ha podido ser creado el fichero");
		} catch (IOException ioe) {
			ioe.printStackTrace();
		 }
		// Crea un objeto File dada la ruta del directorio y el nombre
		//del fichero por separado
		File f2 = new File("C:\\Ficheros", "nuevo2.txt");
		// Crea un objeto File dado el directorio y el nombre
		//del fichero por separado
		try {
		// A partir del objeto File creamos el fichero físicamente
		 if (f2.createNewFile())
			System.out.println("El fichero se ha creado correctamente");
		 else
			System.out.println("No ha podido ser creado el fichero");
		} catch (IOException ioe) {
			ioe.printStackTrace();
		}
		File dir = new File("C:\\Ficheros");
		File f3 = new File(dir, "nuevo3.txt");
		try {
		// A partir del objeto File creamos el fichero físicamente
		if (f3.createNewFile())
			System.out.println("El fichero se ha creado correctamente");
		else
			System.out.println("No ha podido ser creado el fichero");
		 } catch (IOException ioe) {
			ioe.printStackTrace();
		 }

	}
}
```

Ejemplo de creación de un fichero para escritura:

```
import java.io.*;

public class CreaFichero2 {
	public static void main(String [] args){
	String frase="Esto es un ejemplo de escritura de ficheros de texto ";
	try {
		//Crear un objeto File se encarga de crear o abrir acceso a un archivo que
		// se especifica en su constructor
		File archivo=new File("texto.txt");
		// Crear objeto FileWriter que será el que nos ayude a escribir sobre //archivo
		FileWriter escribir=new FileWriter(archivo);
		// Crea el fichero en la carpeta del proyecto
		// FileWriter escribir=new FileWriter(archivo,true); // Para añadir
		//Se escribe en el archivo con el metodo write
		escribir.write(frase);
		// Se cierra la conexión
		escribir.close();
	}
		catch(Exception e){ System.out.println("Error al escribir"); }
	}
}
```

1.2.- Paquete java.nio

En las primeras versiones de Java, el sistema de entrada/salida proporcionado estaba en el **paquete java.io**. En la versión 1.4 de Java se añadió un nuevo sistema de entrada/salida llamado **NIO** para suplir algunas de sus deficiencias que, posteriormente en Java 7 se mejoró aún más con **NIO.2**.

Entre las mejoras, se incluye permitir la navegación sencilla de directorios, soporte para reconocer enlaces simbólicos, leer atributos de ficheros tales como permisos e información sobre el fichero, soporte de entrada/salida asíncrona y soporte para operaciones básicas sobre ficheros como copiar y mover ficheros.

Entre las carencias de la versión anterior que corrige, tenemos:

- **Falta de método de copia:** para copiar un archivo, necesitamos crear dos instancias File y usar un búfer para leer de una y escribir en la otra instancia de File.
- **Mal manejo de errores:** algunos métodos devuelven solo un booleano como indicador de si una operación se ha realizado correctamente o no**.**
- **La API IO bloquea**: nuestro hilo está bloqueado hasta que se complete la operación de E/S. El problema es que cuando un desarrollador intenta leer o escribir algo en un archivo usando Java IO, bloquea el archivo y bloquea el acceso a él hasta que finaliza el trabajo.
- No hay soporte para **enlaces simbólicos**.

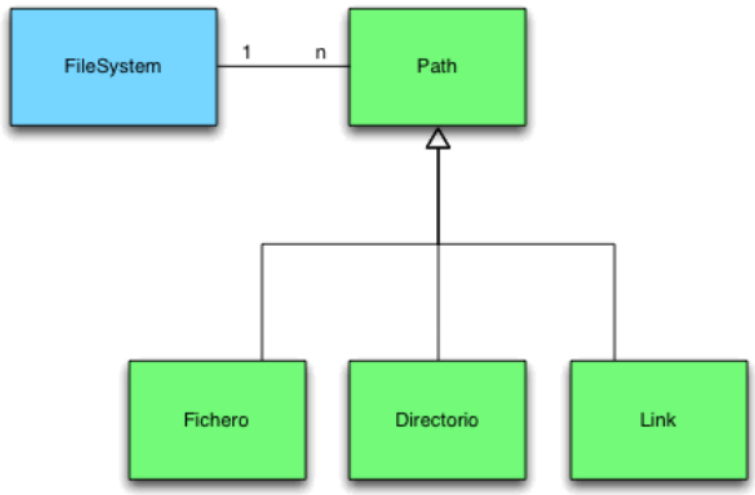
La principal diferencia es que **NIO no bloquea**, mientras que **IO bloquea**. El paquete java.nio puede realizar operaciones de entrada/salida (IO) sin bloqueo. Aquí, sin bloqueo significa que puede leer los datos/información que encuentre listo.
Por ejemplo, tenemos un hilo que le pide al canal que lea los datos/información del búfer; el hilo va para otras operaciones en ese marco de tiempo y continúa de nuevo donde dejó el trabajo. Mientras tanto, el proceso de lectura se completa y aumenta el rendimiento general.

Otra diferencia, es que mientras **IO trabaja con streams**, **NIO opera con canales y búferes**. En NIO, leemos datos desde un búfer en un canal durante la lectura, y escribimos datos desde un búfer al canal durante la escritura.

- **Búfer (Buffer):** Contiene datos primitivos y se utiliza para leer y almacenar datos en caché.
- **Canal (Channel):** Similar a los streams, se utiliza para la comunicación bidireccional.

No vamos a ver casos contratamiento de Buffer y Channel en este curso.

En este gráfico vemos los principales elementos de NIO que vamos a usar:



Las clases principales de esta nueva API para el manejo de rutas, ficheros y operaciones de entrada/salida son las siguientes:

- **Path**: es una abstracción sobre una ruta de un sistema de ficheros. No tiene por qué existir en el sistema de ficheros dicha ruta para poder tener un objeto de este tipo. Puede usarse como reemplazo completo de java.io.File. Los métodos File.toPath() y Path.toFile() ofrecen compatibilidad entre ambas representaciones. Hace referencia a un directorio, fichero o link que tengamos dentro de nuestro sistema de ficheros.

  ```
  public interface Path
  extends Comparable<Path>, Iterable<Path>, Watchable
  ```

- **Files**: es una clase de utilidades con operaciones básicas sobre ficheros. Esta clase consta exclusivamente de métodos estáticos que operan en archivos, directorios u otros tipos de archivos.

  En la mayoría de los casos, los métodos definidos en esta clase delegarán en el proveedor del sistema de archivos asociado la realización de las operaciones con el archivo.

  ```
  public final class Files extends Object
  ```

- **Paths:** Esta clase contiene métodos estáticos que crean objetos Path.

  ```
  public final class Paths extends Object
  ```

- **FileSystem**: Clase de utilidades que proporciona una interfaz para un sistema de ficheros y permite crear objetos para acceder a ficheros y otros objetos del sistema de ficheros.

```
public abstract class FileSystem
extends Object
implements Closeable
```

- **FileSystems:** Actúa como una fábrica para crear nuevos sistemas de ficheros (provee métodos para ello). Esta clase define el método getDefault() para obtener el sistema de archivos predeterminado y los métodos de fábrica para construir otros tipos de sistemas de archivos.

```
public final class FileSystems
extends Object
```

Paths en Java

La interfaz **java.nio.file.Path** representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros.

La forma más sencilla de construir un objeto que cumpla la interfaz Path es a partir de la clase **java.nio.file.Paths**, que tiene métodos estáticos que retornan objetos Path a partir de una representación tipo String del path deseado, por ejemplo:

```
Path p = Paths.get("/home/ejemplos/unFichero");
```

No es necesario que los ficheros existan de verdad en el disco duro para que se puedan crear los objetos Path correspondientes: La representación y manejo de paths en Java no está restringida por la existencia de esos ficheros o directorios en el sistema de ficheros, igual que pasaba con la clase File de java.io.

El interfaz Path declara numerosos métodos que resultan muy útiles para el manejo de rutas(paths), como por ejemplo, obtener el nombre corto de un fichero, obtener el directorio que lo contiene, resolver paths relativos, etc.

Trabajar con paths no tiene nada que ver con trabajar con el contenido de los ficheros que representan, por ejemplo, modificar el contenido de un fichero es una operación que poco tiene que ver con su nombre o su localización en el sistema de ficheros. Una instancia de tipo Path refleja el sistema de nombrado del sistema operativo subyacente, por lo que objetos path de diferentes sistemas operativos no pueden ser comparados fácilmente entre sí.

Ejemplo:

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class _4_EjemploPath1 {
   public static void main(String args[]) {
          Path path = Paths.get("c:\\NUEVODIREC");
          // El directorio no tiene por qué existir

          System.out.println(" path = " + path);
          System.out.println(" is absolute ? = " + path.isAbsolute());
          System.out.println(" file short name = " + path.getFileName());
          System.out.println(" parent = " + path.getParent());
          System.out.println(" uri = " + path.toUri());
          System.out.println(" ruta = " + path);

   }

}
```

Ficheros

La clase **java.nio.file.Files** es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros reales del disco desde Java. Esta clase tiene métodos estáticos para el manejo de ficheros, lo que permite crear y borrar ficheros y directorios, comprobar su existencia en el disco, comprobar sus permisos, moverlos de un directorio a otro y lo más importante, leer y escribir el contenido de dichos ficheros. Veamos cómo se realizan algunas de estas operaciones.

Existencia y comprobación de permisos

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;

public class _2_ComprobarExistencia1 {
   public static void main(String args[]) {
          Path path = Paths.get("test.txt");

          System.out.println(" path = " + path);
          System.out.println(" exists = " + Files.exists(path));
          System.out.println(" readable = " + Files.isReadable(path));
          System.out.println(" writeable = " + Files.isWritable(path));
          System.out.println(" executable = " + Files.isExecutable(path));

   }
}
```

Otra forma:

```
import java.nio.file.Path;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Files;


public class _2_ComprobarExistencia2 {
   public static void main(String args[]) {
          FileSystem sistemaFicheros = FileSystems.getDefault();
          Path rutaFichero = sistemaFicheros.getPath("test.txt");
```

```
        System.out.println(" path = " + rutaFichero);
        System.out.println(" exists = " + Files.exists(rutaFichero));
        System.out.println(" readable =" + Files.isReadable(rutaFichero));
        System.out.println(" writeable =" + Files.isWritable(rutaFichero));
        System.out.println(" executable=" +Files.isExecutable(rutaFichero));

    }
}
```

Creación y borrado

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class _6_CrearFichero1 {
    public static void main(String args[]) {
        Path emptyFile = Paths.get("/examples/emptyFile.txt");
        // tiene que existir la ruta que lo contiene
        //Path emptyFile = Paths.get("emptyFile.txt");

        if (Files.notExists(emptyFile)) {
            try {
                    emptyFile = Files.createFile(emptyFile);
                    System.out.println("Fichero creado");
              } catch (IOException e) {
                    e.printStackTrace();
              }
        }
        else
              System.out.println("Ya existe el fichero");
    }

}
```

Si no existe la ruta contenedora:

```
import java.io.IOException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class _6_CrearFichero2 {
    public static void main(String args[]) {
        try {
              crearFichero("ejemplos/FicheroVacio2.txt");
              System.out.println("Fichero creado en su ruta");
        } catch (FileAlreadyExistsException e) {
              System.out.println("El fichero ya existe");
        } catch (IOException e) {
              System.out.println(e.getMessage());
        } catch (Exception e) {
              System.out.println(e.getMessage());
        }
    }


    public static void crearFichero(String ruta) throws IOException {
        Path RutaContenedora = Paths.get(ruta);
        Files.createDirectories(RutaContenedora.getParent());
        Files.createFile(RutaContenedora);
    }
}
```

Para borrar:

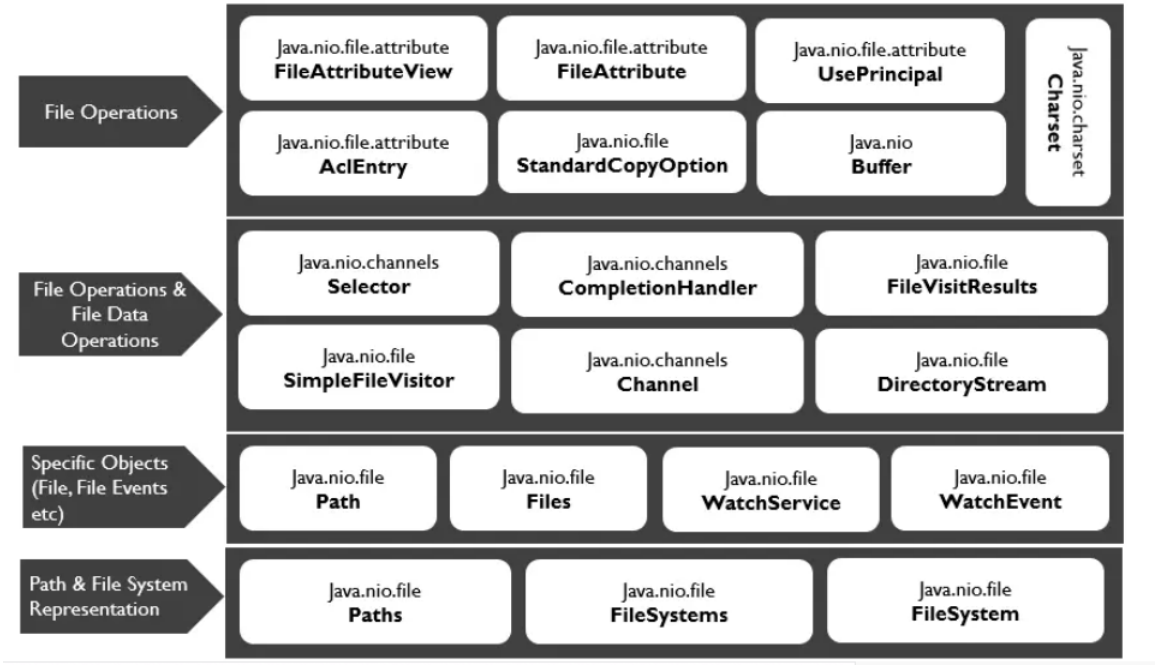
```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Si existe lo borro, si no existe lo creo

public class _5_CreaOBorra {
   public static void main(String args[]) {

          Path path = Paths.get("C:\\Users\\Zeus\\Downloads\\progit.pdf");
          try {
                if (Files.exists(path)) {
                      Files.delete(path);
                      // A diferencia de io, nio produce una excepcion
                      // en caso de fallo
                      System.out.println(" Borrado.");
                } else {
                      Files.createFile(path); // Esto crea un fichero
                      System.out.println(" Creado.");
                }

          } catch (IOException e) {
                System.err.println(e);
                System.exit(1);
          }
   }

}
```

Lectura y escritura

Existe toda una serie de métodos que facilitan la lectura o escritura del contenido de un archivo, los veremos con ejemplos.

Una de las desventajas de `FileInputStream` y `FileOutputStream es` que cuando se crean objetos de estas clases, los archivos se crean inmediatamente en el disco. Y todas las excepciones relacionadas con la creación de archivos podrían generarse potencialmente.

A continuación, se presenta un esquema con los elementos de java.nio:



Veremos en clase varios ejemplos de uso con java.nio. Están en el aula virtual.

2. Operaciones sobre ficheros.

Los **tipos** de operaciones son:

a) Operación de Creación
b) Operación de Apertura
Varios modos:
b.1) Solo lectura
b.2) Solo escritura
b.3) Lectura y escritura
c) Operaciones de lectura / escritura
d) Operaciones de inserción / borrado
e) Operaciones de renombrado / eliminación
f) Operación de desplazamiento dentro de un fichero
g) Operación de cierre

Las **operaciones** para el manejo habitual de un fichero:

1.- Crearlo
2.- Abrirlo
3.- Operar sobre él (lectura/escritura, inserción, borrado, etc.)
4.- Cerrarlo

3. Tipos de ficheros.

La clasificación de los ficheros según el **acceso** a la información almacenada es:

a) **Acceso secuencial**: Para acceder a los datos es necesario pasar por todos los anteriores. **Ej**: Parecido a lo que sucede con una cinta de cassete.
b) **Acceso directo o aleatorio**: Se puede acceder a un dato sin pasar por todos los anteriores. **Ej**: Parecido a lo que sucede con acceso a un disco duro, a arrays en Java...

Clasificación de los ficheros según el **contenido**:

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:

- Los archivos de **caracteres** (o **de texto**). Almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF8, UTF16, etc.).
- Los archivos de **bytes** (o **binarios**). Almacenan secuencias de dígitos binarios (ej: ficheros que almacenan enteros, floats, imágenes…).

Un **fichero de texto** es aquel formado exclusivamente por caracteres y que, por tanto, puede crearse y visualizarse usando un editor (plano). Las operaciones de lectura y escritura trabajarán con caracteres. Por ejemplo, los ficheros con código java son ficheros de texto.

En cambio, un **fichero binario** ya no está formado por caracteres, sino que los bytes que contiene pueden representar otras cosas como números, imágenes, sonido, etc.

4. Conceptos Básicos de Entrada/Salida

En Java se define la abstracción de **stream** (flujo) para tratar la comunicación de información entre el programa y el exterior. Entre una fuente y un destino fluye una secuencia de datos. No confundir con los Streams introducidos con java 8 (los veremos en algún momento).

Los flujos actúan como **interfaz con el dispositivo o clase asociada** y proporcionan una **operación independiente del tipo de datos y del dispositivo**. Nos permiten usar diversidad de dispositivos (fichero, pantalla, teclado, red…) y diversidad de formas de comunicación.

Flujos estándar

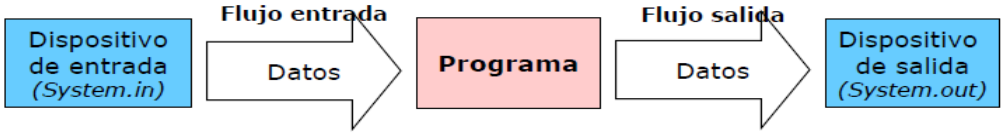
Entrada estándar - habitualmente el teclado
Salida estándar - habitualmente la consola
Salida de error - habitualmente la consola

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

System.in implementa la entrada estándar (InputStream)
System.out implementa la salida estándar (PrintStream)
System.err implementa la salida de error (PrintStream)

La entrada/salida estándar (normalmente el teclado y la pantalla, respectivamente) se define mediante dos objetos que puede usar el programador sin tener que crear flujos específicos.

La clase ***System*** tiene un miembro dato denominado ***in*** que es un objeto de la clase ***InputStream*** que representa al teclado o flujo de entrada estándar. Sin embargo, el miembro ***out*** de la clase *System* es un objeto de la clase ***PrintStream***, que imprime texto en la pantalla (la salida estándar).



Flujos estándar:

System.in (*)

Instancia de la clase *InputStream*: flujo de bytes de entrada

Métodos para la lectura de datos, entre otros:

- *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*: flujo de bytes de salida

Métodos para impresión de datos, entre otros:

- *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)

Nota (*): System.out.println(System.in.getClass()); // vemos que es instancia de una subclase

Para **leer un carácter** solamente tenemos que llamar a la función ***read*** desde ***System.in*** **(**recuerda que System.in que es una instancia de la clase ***InputStream*****)**.

```
try{
        System.in.read();

}catch (IOException ex) {  }
```

Ejemplo:

```
import java.io.*;

//Ejemplo de lectura de líneas de la entrada estándar carácter a carácter
// Contamos los caracteres que hay en una línea.
class LecturaDeLinea {
public static void main( String args[] ) throws IOException {
        int c;
        int contador = 0;
        // Se lee hasta encontrar el fin de línea
        while( (c = System.in.read() ) != '\n' )
        {
                contador++;
                System.out.print( (char) c );
        }
        System.out.println(); // Se escribe el fin de línea
        System.err.println( "Contados "+ contador +" bytes/caracteres en total." );
        }
}
```

Utilización de los flujos:

FLUJOS DE ENTRADA Y SALIDA EN **JAVA**. ... El **flujo** es una secuencia ordenada de datos que tiene una fuente (**flujos** de entrada) o un destino (**flujos** de salida). Los streams soportan varios tipos de datos: bytes simples, tipos de datos primitivos, caracteres localizados, y objetos. Los flujos se implementan en las clases del paquete java.io.
Esencialmente todos funcionan igual, independientemente de la fuente de datos.

Lectura:

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
   - Teclado
   - Fichero
   - Socket remoto
2. Mientras existan datos disponibles:
   - Leer datos
3. Cerrar el flujo (método close())

Escritura:

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
   - Pantalla
   - Fichero
   - Socket local
2. Mientras existan datos disponibles:
   - Escribir datos
3. Cerrar el flujo (método close)

***Nota**: para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos*

Un fallo en cualquier punto produce la excepción IOException, es obligatorio capturarla.

Clasificación de flujos

Según la representación de la información:

Flujos de bytes: clases *InputStream* y *OutputStream*
Flujos de caracteres: clases *Reader* y *Writer*

Recuerda que `System.in` es un *InputStream*

Se puede pasar de un flujo de bytes a uno de caracteres con las clases *InputStreamReader* y *OutputStreamWriter*

Según el propósito:

Entrada: *InputStream, Reader*
Salida: *OutputStream, Writer*
Entrada/Salida: *RandomAccessFile*
Que **realizan algún tipo de procesamiento** sobre los datos (p.e. *buffering*, conversiones, filtrados): *BufferedReader, BufferedWriter, BufferedInputStream*, *BufferedOutputStream…*

Según el tipo de acceso:

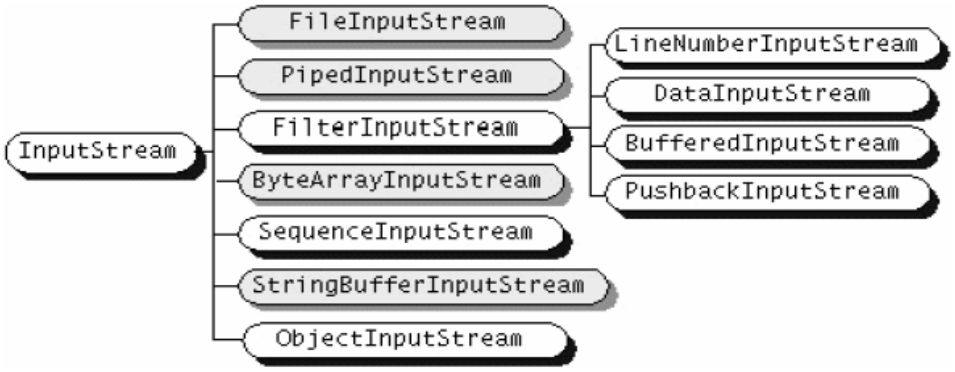
Secuencial (Para acceder a un elemento hay que pasar por todos los demás)
Aleatorio - (*RandomAccessFile*) (Se puede acceder directamente a un elemento)

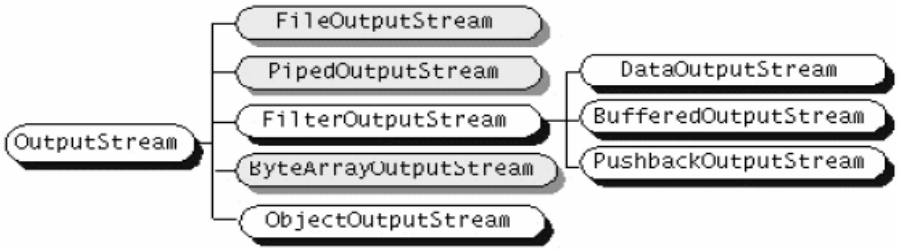
Los flujos están agrupados en el paquete java.io y se dividen en dos jerarquías de clases independientes, una para lectura/escritura binaria (bytes) y otra para lectura/escritura de caracteres de texto (char).

Jerarquía de flujos de bytes (Streams)

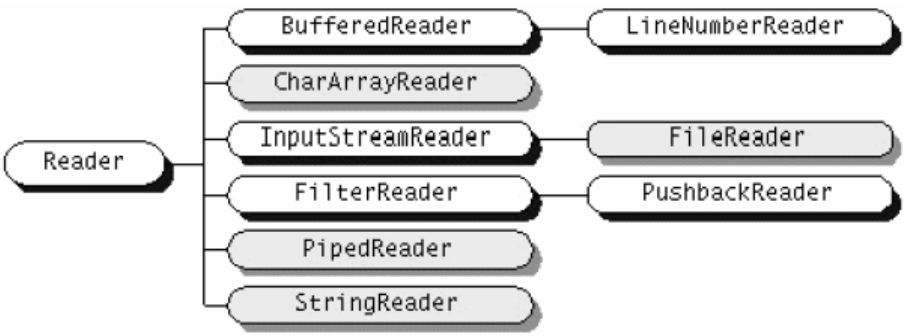
En el siguiente gráfico vemos la relación entre la clase InputStream y sus hijas. Existen para facilitarnos las distintas operaciones que tenemos que hacer. Veremos distintos ejemplos con las más importantes. Recuerda que tratamos con estas clases cuando leamos **bytes o conjuntos de bytes directamente.**

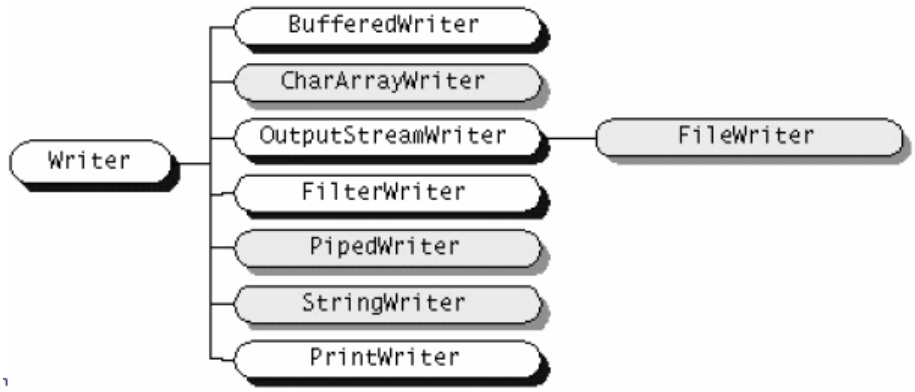
Con estas clases (que tratan con Streams) se podría leer cualquier tipo de fichero, aunque cuando se trate de ficheros de texto puro (de los que se editan en un editor de ficheros planos, como notepad) es más fácil tratarlos con las clases que se muestran en el segundo gráfico (Jerarquía de flujos de caracteres).





Jerarquía de flujos de caracteres





Puedes investigar sobre las principales clases de ambos tipos en la ayuda de Java (API JDK)

Vamos viendo cómo se producen las distintas operaciones de lectura/escritura con las distintas clases, a modo de ejemplos de introducción. Se utilizarán unas u otras según las necesidades de nuestros programas y más adelante en el tema profundizaremos según sean los ficheros de texto o binarios.

Entrada de caracteres (ambas clases son descendientes de la clase Reader, flujo de caracteres)

InputStreamReader

Lee bytes de un flujo InputStream y los convierte en caracteres Unicode.

Métodos de utilidad:

- *read()* lee un único carácter
- *ready()* indica cuando está listo el flujo para lectura

BufferedReader

Entrada mediante búfer, mejora el rendimiento.

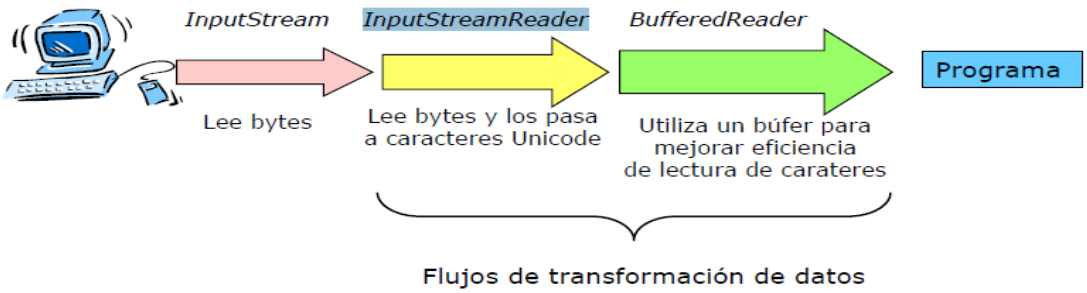
Método de utilidad:

- *readLine()* lectura de una línea como cadena

Ejemplo:

```
InputStreamReader entrada = new InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader (entrada);
String cadena = teclado.readLine();
```

Los flujos se pueden combinar para obtener la funcionalidad deseada:



Ejemplo de combinación de flujos:

```
import java.io.*;
public class Eco {
        public static void main (String[] args) {
                BufferedReader entradaEstandar = new BufferedReader
                (new InputStreamReader(System.in));
                String mensaje;
                System.out.println ("Introducir una línea de texto:");
                mensaje = entradaEstandar.readLine();
                System.out.println ("Introducido: \"" + mensaje + "\"");
        }
}
```

Ejemplo de entrada de texto desde un fichero (para ver cómo usar los Buffered como un adelanto a lo que veremos más adelante):

```
try {
        BufferedReader reader =
        new BufferedReader(new FileReader(NOMBRE_FICHERO));
        String linea = reader.readLine();
        while(linea != null) {
        // procesar el texto de la línea
                linea = reader.readLine();
        }
        reader.close(); // major ponerlo en finally, aquí es peligroso
}
catch(FileNotFoundException e) {
// No se encontró el fichero
}
catch(IOException e) {
// Algo fue mal al leer o cerrar el fichero
}
```

Otro ejemplo (sería lo que viene a hacer la clase Scanner):

```
import java.io.*;

class Teclado {
	/** variable de clase asignada a la entrada estándar del sistema */
	public static BufferedReader entrada =
	new BufferedReader(new InputStreamReader(System.in));

	/** lee una cadena desde la entrada estándar
	* @return cadena de tipo String
	*/
	public static String leerString() {
		String cadena="";

		try {
			cadena = new String(entrada.readLine());
		}catch (IOException e) {
			System.out.println("Error de E/S");
		}
		return cadena;
	}

	/** lee un numero entero desde la entrada estandar
	* @return numero entero de tipo int
	*/
	public static int leerInt() {
		int entero = 0;
		boolean error = false;

		do {
		try {
			error = false;
			entero = Integer.valueOf(entrada.readLine());
		}catch (NumberFormatException e1) {
			error = true;
			System.out.println("Error en el formato del número,
			intentelo de nuevo.");
		}catch (IOException e) {
			System.out.println("Error de E/S");}
		} while (error);

		return entero;
	}
} // final de la clase Teclado
```

5. Ficheros de Texto.

Existen dos clases que manejan caracteres que son **FileWriter** y **FileReader**.

Veamos su jerarquía:

java.lang.Object java.io.Writer java.io.OutputStreamWriter **java.io.FileWriter**	java.lang.Object java.io.Reader java.io.InputStreamReader **java.io.FileReader**

La construcción de objetos del tipo **FileReader** se hace con un parámetro que puede ser un objeto **File** o un String que representarán a un determinado archivo.

La construcción de objetos **FileWriter** se hace igual, aunque también se le puede añadir un **segundo parámetro** booleano, que tomará valor true si se abre el archivo para añadir datos o false para crearlo por primera vez; si tuviese datos estos se borrarían en este segundo caso.

Para escribir se utiliza el método **write** que recibe como parámetro lo que se desea escribir en formato int o String y para leer se utiliza el método read, que devuelve un int, el valor en código ASCII, del carácter leído. En el momento en que se haya alcanzado el **final del fichero** (no haya más datos que leer) el método read retorna el valor -1.

Métodos básicos de **Reader**:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

Métodos básicos de **Writer**:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Ejemplo de lectura de un fichero (ya existente) carácter a carácter:

```
//Ejemplo de lectura de un fichero carácter a carácter.
import java.io.*;
public class LeeFichero{
   public static void main(String arg[]){
        // Se define un int que va a contener los caracteres del archivo
        int c;
        try{
                // Se crea un objeto FileReader que obtiene lo que tenga el archivo
                FileReader lector=new FileReader("texto.txt");
                c=lector.read();
                while(c!=-1){
                        System.out.println((char) c);
                        c=lector.read();
                }
                lector.close();
        }
        catch(Exception e){
                System.out.println("Error al leer");
        }
        }
}
```

Concepto de Buffering

Cualquier operación que implique acceder a almacenamiento externo es muy costosa, por lo que es interesante intentar reducir al máximo las operaciones de lectura/escritura que realizamos sobre los ficheros, haciendo que cada operación lea o escriba muchos caracteres. Además, eso también permite operaciones de más alto nivel, como la de leer una línea completa y devolverla en forma de cadena. De esta forma, se va guardando lo que se ha leído en un buffer (en memoria de nuestro programa) hasta que se opera con ello de una sola vez.

Las clases **BufferedReader** y **BufferedWriter** leen y escriben un texto desde un stream de entrada o de salida. Proporcionan un buffer de almacenamiento temporal.

La **escritura** se realiza con el método **write** que permite grabar caracteres, Strings y arrays de caracteres en el fichero. Permite utilizar el método newLine() que escribe un salto de línea en el archivo.

java.lang.Object
  java.io.Writer
  java.io.BufferedWriter
	java.lang.Object
	  java.io.Reader
	  java.io.BufferedReader

Para **leer**, se tiene el método **readLine**() que lee una línea de texto entera cada vez. En este caso, cuando se haya alcanzado el final del fichero el método retorna el valor null.

No es posible usarlas por sí mismas, se deben envolver en un Reader o en un Writer dentro de un BufferedReader o de un BufferedWriter respectivamente. Esto es así porque sus constructores necesitan un parámetro de tipo Reader y Writer respectivamente. Busca estas clases en la ayuda del jdk y compruébalo.

```
public class LeeFicheroLineas {
   public static void main(String arg[]){
                //Creamos un String que va a ir conteniendo todo el texto del archivo
                String texto="";

                try{
                        // Se crea un objeto FileReader que obtiene lo que tenga el archivo
                        FileReader lector=new FileReader("texto.txt");
                        //El contenido del objeto se envuelve en un BufferedReader
                        BufferedReader contenido=new BufferedReader(lector);
                        texto= contenido.readLine();
                        while(texto!=null){
                                System.out.println(texto);
                                texto= contenido.readLine();
                        }
                        contenido.close(); // Se puede omitir
                        lector.close();
                }catch(Exception e){
                        System.out.println("Error al leer");
                }
                }
        }
```

Otra opción de escritura es usar la clase PrintWriter:

salida =new PrintWriter(new FileWriter(nombre))

También:

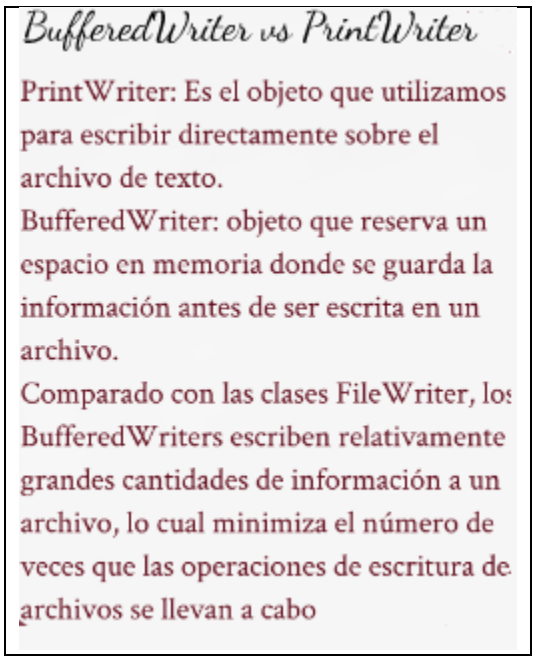
FileWriter (nombre, añadir) // así escribimos al final del fichero

Con los métodos:

println()
print()
close():IOException

Ejemplo:

```
import java.io.*;
public class PruebaEscritura {

   public static void main(String[] args) {

      try {
               FileWriter connection = new FileWriter("C:\\test.txt", true);
               PrintWriter file = new PrintWriter(connection);
               file.println("Hola");
               file.println("Hola");
               file.close();
               writeAgain();
      }

      catch (IOException e) {
         System.out.println("IOException");
      }

   }

   public static void writeAgain() throws IOException {
      FileWriter connection = new FileWriter("C:\\test2.txt", true);
      BufferedWriter file = new BufferedWriter(connection);
      file.write("Adios");
      file.write("Adios");
      file.close();
   }

}
```



- **FileWriter:** es un objeto que tiene como función escribir datos en un archivo.
- **BufferedWriter**: objeto que reserva un espacio en memoria (buffer) donde se guarda la información antes de ser escrita en un archivo.
- **PrintWriter**: Es el objeto que utilizamos para escribir directamente sobre el archivo de texto.

6. Ficheros binarios.

Definición y uso

Un **fichero binario** es un fichero que almacena una secuencia de datos codificados en binario (secuencias de datos short, int, double, boolean, char, etc.)

Para trabajar con este tipo de ficheros, es decir, para escribir o leer en ellos la información binaria que almacenan, se necesita conocer el patrón que guía su disposición en el fichero. En un fichero binario no hay líneas.

Vamos a ver dos formas de usarlos:

FileOutputStream y FileInputStream:

java.lang.Object java.io.OutputStream **java.io. FileOutputStream**	java.lang.Object java.io.InputStream **java.io. FileInputStream**

Es una pareja de clases que nos permite leer grupos de bytes de un tamaño definido, por ejemplo, nos serviría para copiar un fichero de imagen.

Métodos asociados:

void	**write**(byte[] b) Writes b.length bytes from the specified byte array to this file output stream.
void	**write**(byte[] b, int off, int len) Writes len bytes from the specified byte array starting at offset off to this file output stream.
void	**write**(int b) Writes the specified byte to this file output stream.
int	**read**() Reads a byte of data from this input stream.
int	**read**(byte[] b) Reads up to b.length bytes of data from this input stream into an array of by
int	**read**(byte[] b, int off, int len) Reads up to len bytes of data from this input stream into an array of bytes.

Pero como puedes ver en los ejemplos, también valdría para hacer una copia de un fichero de texto:

EjFileInputOutputStream1.java y EjFileInputOutputStream2.java

DataOutputStream y DataInputStream:

java.lang.Object java.io.OutputStream java.io.FilterOutputStream **java.io. DataOutputStream**	java.lang.Object java.io.InputputStream java.io.FilterInputStream **java.io. DataInputStream**

La clase **DataOutputStream** permite crear un objeto que se asocia a un objeto **FileOutputStream** y facilita métodos para escribir o almacenar secuencialmente información codificada en binario en el fichero asociado a dicho objeto. La escritura de un nuevo dato se produce de forma secuencial.

La escritura física de datos en el fichero no se produce uno a uno, sino que se gestiona en bloques mediante un **buffer** o almacén intermedio. El programador debe gestionar la captura y tratamiento de IOException.

Los **métodos** más usados son:

void writeBoolean(boolean val)
void writeChar(int val)
void writeInt(int val)
void writeDouble(double val)
void writeChars(String str)
void writeUTF(String str)

Ej: Crea un fichero binario compuesto por una secuencia de pares (Nmat, nota) solicitados por teclado.

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

public class Binarios1 {
  public static void main(String args[]) {
        int nm;
        double nota;
        Scanner teclado = new Scanner(System.in);
        try {
           FileOutputStream fos=new FileOutputStream("notas.dat");
           DataOutputStream dos= new DataOutputStream(fos);
           System.out.print("Introduzca un NMat (0 para acabar): ");
           nm = teclado.nextInt();
           while (nm != 0) {
                 dos.writeInt(nm);
                 System.out.print("Introduzca una nota: ");
                 nota = teclado.nextDouble();
                 dos.writeDouble(nota);
```

```
                System.out.print("Introduzca un NMat (0 para acabar): ");
                nm = teclado.nextInt();
            }
            dos.close();
            fos.close();
        }
        catch (IOException ex) {
                System.out.println("Error: " + ex.getMessage());
        }
    }
}
```

La clase **DataInputStream** permite crear un objeto que se asocia a un objeto **FileInputStream** y facilita métodos para leer de él secuencialmente información codificada en binario.
La lectura de un nuevo dato se produce de forma secuencial, se lee la secuencia, información almacenada en el fichero detrás de la última información leída.

El programador debe gestionar la captura y tratamiento de IOException. El intento de lectura de un dato cuando ya ha sido leído todo el fichero lanza la excepción EOFException. Su gestión es clave en la programación de operaciones de lectura de un fichero binario.

Los **métodos** más usados son:

```
boolean readBoolean()
byte readByte()
char readChar()
int readInt()
double readDouble()
String readUTF()
```

Ej: Muestra el fichero binario compuesto por una secuencia de pares (NMat, nota)

```
public class Binarios2 {
    public static void main(String args[]) {
        int nm;
        double nota;
        try {
            FileInputStream fis=new FileInputStream("notas.dat");
            DataInputStream dis = new DataInputStream(fis);
            System.out.println(" NMAT   Nota");
            try {
              while (true){
                nm = dis.readInt();
                nota = dis.readDouble();
                System.out.printf("%6d %5.1f \n", nm, nota);
              }
            }catch (EOFException ex) {
                System.out.println("FINAL DE FICHERO " +
                ex.getMessage());
            }
```

```
                dis.close(); // mejor en finally o try con recursos
                fis.close(); // mejor en finally o try con recursos
                }
                catch (IOException ex) {
                        System.out.println("Error: " + ex.getMessage());
                }
        }
}
```

También podríamos haber usado para leer el fichero la condición:

```
while (dis.available() > 0)
```

Evitando leer el fin de fichero

7. Ficheros de acceso aleatorio- acceso directo (binarios).

Hasta ahora los archivos se han leído o escrito secuencialmente, es decir, desde el inicio hasta el final; pero es posible, leer datos o escribir datos en una zona concreta del archivo.
La clase **RandomAccessFile** permite acceder al archivo en forma aleatoria. Es decir, se permite leer o escribir cualquier posición del archivo en cualquier momento. Esta clase implementa las **interfaces DataInput y DataOutput** que sirven para leer y escribir datos.

La construcción requiere de una cadena que contenga un archivo File y un segundo parámetro obligatorio denominado **modo**, que va a especificar el tipo de operación que se realizará sobre el fichero. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

Ej:

```
File f=new File("prueba.txt");
RandomAccessFile archivo = new RandomAccessFile(f, "rw");
```

Como ocurría en las clases anteriores, hay que capturar la excepción. Los métodos fundamentales son:

void seek(long pos). Permite colocar el puntero en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es la señal que marca la posición a leer o escribir.

long getFilePointer(). Posición actual del puntero de acceso

long length(). Devuelve el tamaño del archivo
int skipBytes(int desplazamiento). Desplaza el puntero desde la posición actual, el número de bytes indicado por desplazamiento

Los **métodos de lectura**, leen un dato del tipo indicado y son equivalentes a los disponibles en la clase DataInputStream.

readBoolean, readByte, readInt, readDouble, readFloat, readUTF, readLine.

Los **métodos de escritura** reciben como parámetro el dato a escribir y escriben encima de lo hubiese ya escrito, por lo que para añadir hay que colocar el puntero de acceso al final del archivo.

writeBoolean, writeByte, writeBytes, writeInt, writeDouble, writeFloat, writeUTF, writeLine.

Ej: Crear un fichero binario, de acceso aleatorio, con los 100 primeros números y mostrarlo en pantalla.

```
import java.io.*;
public class Aleatorio{
	public static void main(String args[]) {
		File f = new File("prueba.dat");
		RandomAccessFile raf=null;
		try {
			raf = new RandomAccessFile(f,"rw");
			for ( int i=1; i <= 100 ; i++ )
				raf.writeInt( i );
			try{
				System.out.println( " El fichero ocupa " + raf.length() +
				 " bytes." );
				raf.seek(0); // La primera posición empieza en 0
				System.out.print(" En la posicion " +
				raf.getFilePointer());
				System.out.println(" está el número " + raf.readInt() );
				raf.skipBytes( 9*4 ); // Salto 9 => Elemento 10 más allá
				System.out.print(" 10 elementos más allá, está el ");
				System.out.println(raf.readInt());
				raf.close();
			}
			catch(IOException e){
				System.out.println();
			}
		}catch (FileNotFoundException e) {
			System.out.println();
		}
		catch(IOException e){
			System.out.println();
		}
	}
}
```

Ej: Crea un fichero con los 10 primeros números enteros y después modificalo multiplicando por 2 las componentes pares (valores escritos en el fichero).

```
import java.io.*;
public class Aleatorios2{
	public static void main(String args[]) {
		File f = new File("prueba.dat");
		RandomAccessFile raf=null;
		try {
			raf = new RandomAccessFile(f,"rw");
			for ( int i=1; i <= 10 ; i++ )
				raf.writeInt( i );
			try{
				raf.seek(0); // La primera posición empieza en 0
				while(true){
					int valor=raf.readInt();
					System.out.println(valor);
					if(valor%2==0){
						valor=valor*2;
				System.out.println("Escribo "+valor+
						" En la posicion " +
						((raf.getFilePointer()-4)/4 + 1));
						raf.seek(raf.getFilePointer()-4);
						//Me posiciono 4 bytes atrás
						raf.writeInt(valor);
					}
				}
			}catch(EOFException e){

			}


		}
		catch(IOException e){
			System.out.println(e);
		} finally {
	try {
	   if (raf != null) {
	      raf.close();
	   }
	} catch (IOException e) {
	   System.out.println(e.getMessage());
	}
	}
	}
}
```

8. Serialización.

Un objeto **serializable** es un objeto que se puede convertir en una secuencia de bytes. Es una forma automática de guardar y cargar el estado de un objeto.

Se basa en la interfaz Serializable (en el paquete java.io) que es la que permite esta operación. Si una clase implementa esta interfaz, sus objetos pueden ser guardados y restaurados directamente en un archivo.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, la clase debe implementar la interfaz Serializable, incluyendo en la cabecera de la clase **implements Serializable.** Esta interfaz no posee métodos, pero es un requisito obligado para hacer que un objeto sea serializable.

Las clases ObjectInputStream y ObjectOutputStream se encargan de realizar los procesos de lectura o escritura del objeto de o a un fichero. Son herederas de InputStream y OutputStream (son ficheros binarios los que se generan) y son casi iguales a DataInputStream/DataOutputStream solo que añaden los métodos **readObject** y **writeObject** que son los que permiten utilizar directamente los objetos.

java.lang.Object java.io.OutputStream **java.io. ObjectOutputStream**	java.lang.Object java.io.InputStream **java.io. ObjectInputStream**

Escribir objetos al flujo de salida **ObjectOutputStream** es muy simple y requiere los siguientes **pasos**:

a) Se crea el objeto de la clase serializable.
b) Se crea un objeto de la clase FileOutputStream, con el nombre de fichero y opcionalmente, se le puede añadir el valor true si en el fichero se van a añadir datos.
c) Se vincula el objeto de la clase FileOutputStream al del flujo de salida de la clase ObjectOutputStream que es que va a procesar los datos.

Ej:

```
FileOutputStream fos        =new FileOutputStream("fichero.dat");
ObjectOutputStream oos      =new ObjectOutputStream(fos);
```

d) Con el método writeObject se escriben los objetos al flujo de salida y los guarda en el archivo.
e) Finalmente hay que cerrar los flujos.

Ejemplo:

```
import java.io.*;
public class Persona implements Serializable{
  String nombre;
  int edad;
  public Persona(String n,int e) {
        nombre=n;
        edad=e;
  }

void CambiarEdad(int e){
        edad=e;
  }
  void Escribir(){
        System.out.println("Nombre "+nombre+" Edad "+edad);
  }
}
```

El **proceso de lectura** es similar al proceso de escritura y es muy simple. Los **pasos** a seguir son los siguientes:

a) Se crea un flujo de entrada a disco, pasándole el nombre del archivo.
b) El flujo de entrada ObjectInputStream es el que procesa los datos y se vincula a un objeto de la clase FileInputStream.
c) El método readObject lee los objetos del flujo de entrada, en el mismo orden en el que ha sido escritos. Es necesario hacer un cast al tipo de la clase. Cuando no haya más datos que leer saltará la excepción EOFException.
d) Finalmente, se cierra los flujos

Ejemplo:

```
import java.io.*;
public class PruebaPersona {
        public static void main(String[] args) {
                Persona p1=new Persona("Pablo", 20);
                Persona p2=new Persona("Rosa", 35);
                try {
                        FileOutputStream fos= new FileOutputStream("datos.obj");
                        ObjectOutputStream oos=new ObjectOutputStream(fos);
                        oos.writeObject(p1);
                        oos.writeObject(p2);
                        oos.close();
                        fos.close();
                }
                catch (IOException ex) {
                        System.out.println(ex);
                }
                try{
                        FileInputStream fis= new FileInputStream("datos.obj");
                        ObjectInputStream ois=new ObjectInputStream(fis);
```

```
                        try{
                                while(true){
                                        p1=(Persona)ois.readObject();
                                        System.out.println("Nombre "+p1.nombre+
                                        " Edad "+p1.edad);
                                }
                        } catch(EOFException e){ }
                        ois.close();
                        fis.close();
                } catch (IOException ex) {
                        System.out.println(ex);
                } catch (ClassNotFoundException ex) {
                        System.out.println(ex);
                }
        }
}
```

Cuando un atributo de una clase contiene información que no se desea que se pueda guardar, hay técnicas para protegerla.
Cuando dicha información es privada, el atributo tiene el modificador private, pero una vez que se ha enviado al flujo de salida cualquiera puede leerla en el archivo en disco o interceptarla en la red. El modo más simple de proteger la información sensible, como una contraseña es la de poner el modificador **transient** delante del atributo que la guarda.

La *salida* del programa es:

Nombre: Pepe
Cuenta: No disponible

Lo que indica que la información que contiene el atributo Cuenta y que tiene el modificador transient no se ha guardado en el archivo. En la reconstrucción del objeto con la información guardada en el archivo el dato Cuenta toma el valor null.

Cuando se quieren añadir registros al fichero surge un problema, pues cada vez que se comienza a escribir se añade una cabecera, por lo que al leer posteriormente los registros salta un error cuando encuentra esas cabeceras. Para solucionar este problema, tenemos dos posibilidades:

1.- Crear un fichero auxiliar en el que se copien todos los registros que se tienen y a continuación los que se quieren añadir. Posteriormente, se borrará el primer fichero y el auxiliar se renombrará con el nombre del primer fichero.

2.- Crear una nueva clase que herede de la clase ObjectOuputStream y que el método writeStreamHeader() lo sobrescriba dejando en blanco.

Ej: Usando la 1ª manera:

```
import java.io.*;

public class PruebaPersona2 {
	public static void main(String[] args) {
		Persona p1;
		try{
			//Se crea el fichero con dos registros
			FileOutputStream fos= new FileOutputStream("datos.obj");
			ObjectOutputStream oos=new ObjectOutputStream(fos);
			p1=new Persona("Pepe",40);
			oos.writeObject(p1);
			p1=new Persona("Carmen",15);
			oos.writeObject(p1);
			oos.close();
			fos.close();
		}catch (IOException ex) {
			System.out.println(ex);
		}

		try{
			//Se añaden dos registros más usando el primer método.
			File f=new File("auxiliar.obj");
			FileOutputStream fos= new FileOutputStream(f);
			ObjectOutputStream oos=new ObjectOutputStream(fos);
			File f1=new File("datos.obj");
			FileInputStream fis= new FileInputStream(f1);
			ObjectInputStream ois=new ObjectInputStream(fis);
			try{
				while(true){
					p1=(Persona)ois.readObject();
					oos.writeObject(p1);
				}
			}
			catch(EOFException e){ }
			catch(ClassNotFoundException e){ }
			// Hemos leído los objetos del fichero datos.obj y los hemos pasado
			// al nuevo fichero temporal
			//Añadimos dos nuevos objetos

			p1=new Persona("Juan",50);
			oos.writeObject(p1);
			p1=new Persona("Maria",35);
			oos.writeObject(p1);

			ois.close();
			fis.close();
			oos.close();
			fos.close();
```

```
                f1.delete();
                f.renameTo(f1);   // auxiliar pasa a ser el fichero datos.obj
            }catch (IOException ex) {
                System.out.println(ex);
            }

            //Ahora leemos el fichero con todos los datos
            try{
                FileInputStream fis= new FileInputStream("datos.obj");
                ObjectInputStream ois=new ObjectInputStream(fis);
                try{
                    while(true){
                        p1=(Persona)ois.readObject();
                        System.out.println("Nombre "+p1.nombre+
                        " Edad "+p1.edad);
                    }
                }
                catch(EOFException e){}
                ois.close();
                fis.close();
            }
            catch (IOException ex) {
                System.out.println(ex);
            }
            catch (ClassNotFoundException ex) {
                System.out.println(ex);
            }
        }
}
```

Ej: A ese fichero que ya tiene 4 registros se le van a añadir 2 más usando la 2ª manera.

```
import java.io.*;

public class PruebaPersona3{
            public static void main(String[] args){
                Persona p1;
                try{
                    // Se añaden dos registros más usando el segundo método.
                    // El fichero tiene que existir previamente para que esto
                    // funcione
                       File f=new File("datos.obj");
                       ClaseOutput co;

                       FileOutputStream fos= new FileOutputStream(f,true);
                       co=new ClaseOutput(fos);
                       p1=new Persona("Pepi",53);
                       co.writeObject(p1);
                       p1=new Persona("Julia",17);
                       co.writeObject(p1);
```

```
                co.close();
                fos.close();
            } catch (IOException ex) {
                System.out.println(ex);
            }

            try{
                //Se lee el fichero con todos los datos
                FileInputStream fis= new FileInputStream("datos.obj");
                ObjectInputStream ois=new ObjectInputStream(fis);
                try{
                    while(true){
                        p1=(Persona) ois.readObject();
                        if(p1 instanceof Persona){
                            p1.Escribir();
                        }
                    }
                 }catch(EOFException e){}
                 ois.close();
                 fis.close();
            }catch (IOException ex){
                    System.out.println(ex);
            }catch (ClassNotFoundException ex){
                    System.out.println(ex);
            }
        }
}
```

Primero se crea la clase que hereda de ObjectOutputStream y que reescribirá el método writeStreamHeader() dejándolo en blanco. Será la clase que se use para escribir en el fichero.
¿Qué haríamos para que el segundo fichero funcione tanto con un fichero vacío como un fichero con datos ya introducidos?

Recapitulando:

Flujos de bytes especiales

File streams:
Para escribir y leer datos en ficheros

Filter streams:
Permiten filtrar datos mientras se escriben o leen (Se construyen sobre otro flujo)
Permiten manipular tipos de datos primitivos.
Implementan las interfaces DataInput y DataOutput y pueden heredar de las clases FilterInputStream yFilterOutputStream.
El mejor ejemplo son las clases DataInputStream y DataOutputStream para leer y escribir datos de tipos básicos

Object streams:
Para escribir y leer objetos
Implementa lo que se denomina ***serialización de objetos*** (guardar un objeto con una representación de bytes).

Uso de filter streams

Para **leer** tipos de **datos primitivos**
Se puede utilizar un *DataInputStream*

```
FileInputStream ficheroEntrada = new FileInputStream("precios.cat");
DataInputStream entrada = new DataInputStream(ficheroEntrada);
double precio= entrada .readDouble();
entrada.close();
```

Para **escribir** tipos **de datos primitivos**
Se puede utilizar un *DataOutputStream*

```
FileOutputStream ficheroSalida = new FileOutputStream("precios.cat");
DataOutputStream salida = new DataOutputStream(ficheroSalida);
salida.writeDouble(234.56);
salida.flush(); //Fuerza la escritura de datos
salida.close();
```

Métodos básicos de **InputStream**:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

Métodos básicos de **OutputStream**:

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

Los Streams se abren automáticamente al crearlos, pero es necesario cerrarlos explícitamente llamando al método **close()** cuando se dejan de usar.

PrintStream / PrintWriter se utilizan para escribir cadenas de texto.

DataInputStream / DataOutputStream se utilizan para escribir/leer tipos básicos (int, long, float,…).

Según el acceso a ficheros se utilizan unas clases u otras.

a) Acceso **Secuencial**: El más común. Puede ser:
    a.1) Acceso **binario**: FileInputStream / FileOutputStream
    a.2) Acceso a **caracteres** (texto): FileReader / FileWriter
b) Acceso **Aleatorio**: Se utiliza la clase RandomAccessFile