UD 1. Acceso a ficheros, flujos, serialización de objetos, ficheros JSON y XML

## Introduccion

En esta unidad estudiaremos:

* [**Java I/O**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/package-summary.html): <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/package-summary.html>
* [**Java NIO.2**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/nio/package-summary.html): <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/nio/package-summary.html>
* **JSON con Java (con la biblioteca** [**GSON**](https://github.com/google/gson) **y una introducción de** [**Moshi**](https://github.com/square/moshi)**)**: <https://github.com/google/gson>, <https://github.com/square/moshi>
* **XML con Java**: **procesadores DOM y SAX, las clases específicas para el tratamiento de la información contenida en un fichero XML, las clases específicas para la vinculación de objetos, las bibliotecas para conversión de documentos XML a otros formatos**.

Gestión de información almacenada en **ficheros, flujos, haciendo especial hincapié en los formatos JSON y algo de XML** mediante aplicaciones informáticas escritas en Java.

a) **Gestión de flujos, ficheros secuenciales, Acceso Directo y Directorios**: **desarrollo de aplicaciones que gestionan información almacenada en ficheros secuenciales, de acceso directo y en el sistema de directorios.** En ella se aprenderá a identificar y utilizar las clases específicas para operar con cada tipo de fichero y con el sistema de directorios y a manejar las excepciones para el tratamiento de los posibles errores.

b) **Gestión de ficheros JSON y, en menor medida, XML**: desarrollo de aplicaciones que gestionan información almacenada en **ficheros JSON (con biblioteca** [**Gson**](https://github.com/google/gson)**) y una introducción a** [**Moshi**](https://github.com/square/moshi). También veremos algo de XML, y prenderemos a utilizar los **procesadores DOM y SAX, las clases específicas para el tratamiento de la información contenida en un fichero XML**, las clases específicas para la vinculación de objetos, las bibliotecas para conversión de documentos XML a otros formatos y a manejar las excepciones para el tratamiento de los posibles errores.

# UD 01.01. Java IO. ficheros y flujos

En este apartado estudiaremos las principales clases y métodos de la API de Java para el acceso a ficheros y flujos de datos:

* [**Java I/O**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/package-summary.html): <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/package-summary.html>

El API Java IO proporciona **clases para entrada y salida a través de flujos de datos, serialización y sistemas de ficheros** (leer y escribir datos en archivos, así como para leer y escribir datos en la consola).

En este apartado estudiaremos **cómo se organizan los archivos y directorios en un sistema de archivos y cómo acceder a ellos con la clase java.io.File** (el modo tradicional de hacerlo).

Luego veremos **cómo leer y escribir datos de archivo con las clases de flujo** (Streams IO, no confundir con la [API Streams](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/stream/package-summary.html)).

Concluimos **discutiendo formas de leer la entrada del usuario en tiempo de ejecución utilizando la clase Console**.

* Lectura y escritura de datos por consola y archivos, empleando flujos de I/O (modo “tradicional”)
* Uso de flujos de E/S para la lectura y escritura de archivos.
* Lectura y escritura de objetos por medio de serialización (Serializable)

## **01.00 JAVA IO. INTRODUCCIÓN.**

* [1. Java I/O](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#1-java-io)
  + [2. Archivos y directorios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#2-archivos-y-directorios)
    - [2.1. Sistema de Archivos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#21-sistema-de-archivos)
      * [Directorio raíz](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#directorio-ra%C3%ADz)
      * [Rutas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#rutas)
    - [2.1. Almacenar Datos como Bytes](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0100javaiointro/#21-almacenar-datos-como-bytes)

### **1. Java I/O**

Las aplicaciones Java, ¿qué pueden hacer fuera del ámbito de gestionar objetos y atributos en la memoria? ¡Al cerrar el programa se pierde todo! **¿Cómo pueden guardar datos para que la información no se pierda cada vez que el programa se termina? ¡Usar archivos, por supuesto!**, es la primera opcion (*o cualquier sistema de persistencia más avanzado, como bases de datos, que abordaremos en la siguiente unidad*).

Se pueden realizar **programas sencillos que guarden el estado actual de una aplicación en un archivo cada vez que la aplicación se cierra y luego cargue los datos cuando se ejecute la aplicación la próxima vez**. De esta manera, la información se preserva entre ejecuciones del programa. Es lo que se denomina, **persistencia**.

Este apartado estudiaremos el **API java.io para interactuar con archivos y flujos**. Comenzamos describiendo **cómo se organizan los archivos y directorios en un sistema de archivos** y mostramos cómo acceder a ellos con la clase [**java.io.File**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html) (el modo tradicional de hacerlo). Luego veremos **cómo leer y escribir datos de archivo con las clases de flujo** (Streams IO, no confundir con la [API Streams](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/stream/package-summary.html)). Concluimos discutiendo formas de leer la entrada del usuario en tiempo de ejecución utilizando la clase **Console**.

En el siguiente apartado, dedicado a **“Java NIO.2”, veremos cómo Java proporciona técnicas más poderosas (y rápidas) para gestionar archivos**.

#### **1.2. Archivos y directorios**

Comenzamos este apartado **repasando** **qué es un archivo y un directorio en un sistema de archivos**. También presentamos la clase [java.io.File](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html) y veremos **cómo usarla para leer y escribir información de archivos**.

##### **1.2.1. Sistema de Archivos**

Para empezar es necesario saber **qué es un sistema de archivos**. Los datos se almacenan en dispositivos de almacenamiento persistentes, **como discos duros o tarjetas de memoria**, por ejemplo.

Un ***archivo*** es un registro dentro del dispositivo de almacenamiento que **contiene datos**.

Los archivos se organizan en ***jerarquías*** utilizando directorios.

Un ***directorio*** es una **ubicación que puede contener archivos y otros directorios**.

Cuando trabajamos con **directorios en Java, a menudo los tratamos como archivos**. De hecho, **se usan muchas de las mismas clases para operar en archivos y directorios**. Por ejemplo, un archivo y un directorio pueden renombrarse con el mismo método de Java.

Para interactuar con archivos, necesitamos conectarnos al ***sistema de archivos***. El *sistema de archivos* se encarga de **leer y escribir datos en un *ordenador***. Los diferentes sistemas operativos utilizan sistemas de archivos diferentes para gestionar sus datos. Por ejemplo, los sistemas basados en Windows usan un sistema de archivos diferente que los basados en Unix (Linux, …). La JVM se conectará automáticamente al sistema de archivos local, lo que te permite realizar las mismas operaciones en múltiples plataformas.

###### **Directorio raíz**

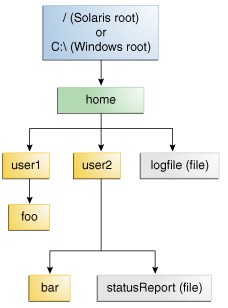
El ***directorio raíz* (root) es el directorio superior en el sistema de archivos, del cual heredan todos los archivos y directorios**:

* En **Windows**, se denota con una **letra de unidad, como c:\\**.
* En **Linux se denota con una barra diagonal simple, /**.

###### **Rutas**

Una ***ruta*** es una **representación en cadena de un archivo o directorio dentro de un sistema de archivos**. Cada sistema de archivos define su propio carácter separador de rutas que se utiliza entre las entradas de directorio. El valor a la izquierda de un separador es el padre del valor a la derecha del separador. Por ejemplo, el valor de ruta /home/otto/cole.txt significa que el archivo cole.txt está dentro del directorio otto, con el directorio otto dentro del directorio home.

***Las rutas pueden ser absolutas o relativas***.

Rerefencia: <https://docs.oracle.com/javase/tutorial/essential/io/path.html>

En la figura anterior muestra un árbol de directorios de ejemplo que contiene un único nodo raíz. Microsoft Windows admite varios nodos raíz. La familia de sistemas operativos basados en Unix (Linux, Solaris, macOS, etc.) admite un único nodo raíz, que se indica mediante el carácter de barra diagonal.

Un archivo se identifica por su ruta en el sistema de ficheros, empezando por el nodo raíz. Por ejemplo, en el sistema de ficheros de Windows, la ruta C:\Programas\holamundo.kt identifica un archivo llamado holamundo.kt que se encuentra en el directorio Programas en la unidad C:.

En la figura: /home/sally/statusReport y c:\home\sally\statusReport son rutas absolutas pa SO Unix y Windows, respectivamente.

El delimitador es específico del sistema de archivos. En Linux \ y en Windows /.

##### **1.2.2. Almacenar Datos como Bytes**

Los datos se almacenan en un sistema de archivos (y en la memoria) **como un 0 o 1, llamado *bit***. Dado que es realmente difícil para las personas leer/escribir datos que son sólo 0s y 1s, **se agrupan en un conjunto de 8 bits, llamado *byte***.

¿Qué pasa con el tipo primitivo byte de Java? Como veremos en el apartado de flujos de E/S, **a menudo se leen o escriben valores en flujos utilizando valores de byte y arrays de bytes**, si bien los métodos recogerán valores enteros para el control de fin de flujo o lectura/escritura.

Caracteres ASCII

Usando un poco de aritmética (2^8), vemos que un byte se puede establecer en uno de 256 posibles permutaciones. Estos **256 valores forman el alfabeto básico** del Sistema Informático para poder escribir caracteres como a, # y 7. Históricamente, **los 256 caracteres se conocen como caracteres ASCII**, basado en el estándar de codificación que los definió. Teniendo en cuenta todos los idiomas (como galego e castelán) y emojis disponibles hoy en día, **256 caracteres es realmente restrictivo**. Se han desarrollado muchos estándares más nuevos que se basan en bytes adicionales para mostrar caracteres.

## **01.01 LA CLASE FILE**

* [1. Clases para trabajar con ficheros (java.io.File, RandomAccessFile, …)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#1-clases-para-trabajar-con-ficheros-javaiofile-randomaccessfile-)
  + [La clase File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#la-clase-file)
* [2. Creación de un Objeto File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#2-creaci%C3%B3n-de-un-objeto-file)
  + [Constructores de la clase File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#constructores-de-la-clase-file)
  + [Campos de la clase File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#campos-de-la-clase-file)
* [3. El objeto *File* vs. archivo real existente](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#3-el-objeto-file-vs-archivo-real-existente)
* [4. Trabajando con un Objeto File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#4-trabajando-con-un-objeto-file)
* [5. Métodos más importantes de java.io.File](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#5-m%C3%A9todos-m%C3%A1s-importantes-de-javaiofile)
  + [Ejercicios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#ejercicios)
* [6. Nuevas características del paquete java.nio.file](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#6-nuevas-caracter%C3%ADsticas-del-paquete-javaniofile)
  + [Conversión entre java.io.File y java.nio.file.Path](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#conversi%C3%B3n-entre-javaiofile-y-javaniofilepath)
  + [Mapeo de la Funcionalidad de java.io.File a java.nio.file](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#mapeo-de-la-funcionalidad-de-javaiofile-a-javaniofile)
* [7. La clase java.io.RandomAccessFile](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#7-la-clase-javaiorandomaccessfile)
  + [Escritura con RandomAccessFile](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#escritura-con-randomaccessfile)
  + [Ejercicios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0101javaiofile/#ejercicios-1)

### **1. Clases para trabajar con ficheros (java.io.File, RandomAccessFile, …)**

Los **flujos de entrada/salida (streams I/O)**, que veremos en esta unidad, **trabajan con gran variedad de fuentes de datos, incluyendo archivos**, sin embargo, **los flujos no proporcionan todas las operaciones comunes a los archivos de disco**.

Existen **clases de E/S para trabajar con ficheros que no son orientas a flujos**. Algunas de ellas son:

* [**java.io.File**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html): ayuda a escribir código independiente de plataforma para examinar y manipular archivos y directorios. Esta clase **era el mecanismo utilizado para E/S de archivos en Java antes de Java 7**: <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html>
* [***java.io.RandomAccessFile***](https://docs.oracle.com/en/java/javase/22/docs/api/index.html): proporciona acceso aleatorio a archivos.
* [**java.nio.file.Path**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/nio/file/Path.html): **interface** añadida en Java 7 y que permite una forma de trabajar con rutas de archivos y directorios más eficiente. Esta interfaz **se emplea con la clase Files para proporcionar un uso más eficiente y completo para acceder a operaciones adicionales**, como atributos de archivos, o excepciones de E/S que ayudan a diagnosticar problemas de E/S.
* [**java.nio.file.Files**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/nio/file/Files.html): **clase** dispone de **métodos estáticos para operaciones de archivos y directorios**, así como **creación de flujos de entrada/salida**.

#### **La clase File**

La primera clase que estudiaremos es **una de las más empleadas (y antigua) del API *java.io***: la clase [**java.io.File**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html).

[La clase File](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html) se utiliza para **leer información sobre archivos y directorios existentes, listar el contenido de un directorio o crear/eliminar archivos y directorios**.

Una **instancia de una clase File** representa la **ruta a un archivo o directorio específico en el sistema de archivos**, pero **no contiene los datos del archivo o directorio** (el archivo podría no existir).

La clase **File no puede leer ni escribir datos dentro de un archivo, aunque se puede pasar como referencia a muchas clases de flujos (y métodos) para leer o escribir datos**. Para escribir leer datos de un archivo, se utilizan las clases de flujo de E/S: FileInputStream, FileOutputStream, FileReader, FileWriter, RandomAccessFile, etc.

Por ello, se usa para **convertir el nombre de un archivo y pasarlo como parámetro a otros métodos o constructores que sí pueden leer o escribir datos**.

FileChannel

La clase FileChannel, del API Java NIO, de java.nio.channels proporciona una forma más avanzada de trabajar con archivos que RandomAccessFile. Tanto File como FileChannel funcionan, pero para **trabajar con puro Java NIO debe usarse la clase FileChannel**.

java.nio.file.Files

**La** [**clase java.nio.file.Files**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/nio/file/Files.html) **proporciona únicamente métodos estáticos para operaciones de archivos y directorios**, así como creación de **flujos de entrada/salida**. Es más eficiente que la clase File y se recomienda su uso en lugar de File **para nuevas aplicaciones.**

### **2. Creación de un Objeto File**

Un objeto File a menudo **se inicializa con una cadena que contiene una ruta absoluta o relativa al archivo o directorio en el sistema de archivos**.

La **ruta absoluta** de un archivo o directorio es la **ruta completa desde el directorio raíz hasta el archivo o directorio**, incluyendo todos los subdirectorios que contienen el archivo o directorio.

La **ruta relativa** de un archivo o directorio es la **ruta desde el directorio de trabajo actual hasta el archivo o directorio**. Por ejemplo, lo siguiente es una ruta absoluta al archivo javaio.txt:

/home/otto/apuntes/javaio.txt

Lo siguiente es una ruta relativa al mismo archivo, asumiendo que el directorio actual del usuario está configurado en /home/otto:

apuntes/javaio.txt

*Los sistemas operativos diferentes varían en su formato de nombres de ruta. Por ejemplo,* ***los sistemas basados en Unix usan la barra diagonal hacia adelante, /, para las rutas, mientras que los sistemas basados en Windows usan el carácter diagonal inversa, \, como separador de ruta****.*

Muchos lenguajes de programación y sistemas de archivos admiten ambos tipos de barras al escribir declaraciones de ruta. Por conveniencia, **Java ofrece dos opciones para recuperar el carácter separador local: una** [**propiedad del sistema**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/System.html#getProperty(java.lang.String)) **y una** [**variable estática**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html#separator) **definida en la clase File**. Ambos ejemplos imprimirán el carácter separador para el entorno actual:

System.out.println(System.getProperty("file.separator"));  
System.out.println(java.io.File.separator);

Prueba el separador de la plataforma (Gitlab) con el siguiente código:

public class HolaMundo {  
 public static void main(String[] args) {  
 System.out.println("Sistema: " + System.getProperty("file.separator"));  
 System.out.println("Atributo: " + java.io.File.separator);  
 }  
}

**Salida:**

El siguiente código **crea un objeto *File*** y determina si la ruta a la que hace referencia el archivo existe en el sistema de archivos:

File javaFile = new File("/home/otto/apuntes/javaio.txt");  
System.out.println(javaFile.exists()); // true, si el archivo existe

Este ejemplo proporciona la ruta absoluta a un archivo y muestra true o false según si el archivo existe.

Tiene **cuatro constructores**:

public File(String pathname)  
public File(File parent, String child)  
public File(String parent, String child)  
public File(URI uri)

El primero crea un objeto File a partir de una ruta en forma de cadena. Los otros dos constructores se utilizan para crear un objeto File a partir de una ruta principal y una secundaria, como la siguiente:

File apuntesJavaIO = new File("/home/otto", "apuntes/javaio.md");  
File directorioPadre = new File("/home/otto");  
File arquivo3 = new File(directorioPadre, "apuntes/javaio.md");

En este ejemplo, creamos dos nuevas instancias de File que son equivalentes la instancia anterior de apuntesJavaIO. **Si la instancia principal es nula, se omitiría y el método volvería al constructor de cadena única**.

#### **Constructores de la clase File**

|  |  |
| --- | --- |
| **Constructor** | **Descripción** |
| File(String pathname) | Crea un objeto File a partir de una ruta en forma de cadena. |
| File(File parent, String child) | Crea un objeto File a partir de una ruta principal y una secundaria. |
| File(String parent, String child) | Crea un objeto File a partir de una ruta principal y una secundaria. |
| File(URI uri) | Crea un objeto File a partir de un URI. |

#### **Campos de la clase File**

La clase File tiene varios campos que puedes usar para acceder a información sobre el sistema de archivos subyacente. Algunos de los campos más útiles son:

|  |  |
| --- | --- |
| **Campo** | **Descripción** |
| static String pathSeparator | El separador de PATH de la plataforma. Por ejemplo, en Windows es ; y en Unix es :. |
| static char pathSeparatorChar | El separador de ruta de la plataforma como un carácter. |
| static String separator | El separador de ruta de la plataforma. Por ejemplo, en Windows es \ y en Unix es /. |
| static char separatorChar | El separador de ruta de la plataforma como un carácter. |

### **3. El objeto File vs. archivo real existente**

Al trabajar con una instancia de la clase File, ten en cuenta que **sólo representa una ruta a un archivo**. A menos que se opere sobre él, **no está conectado a un archivo real en el sistema de archivos**.

Por ejemplo:

* Se puede crear un nuevo objeto File para **comprobar si un archivo existe en el sistema**.
* Se puede llamar a varios métodos para **leer propiedades de archivos dentro del sistema de archivos**.
* Tiene **hay métodos para modificar el nombre o la ubicación de un archivo, así como para eliminarlo**.

La JVM y el sistema de archivos subyacente leerán o modificarán el archivo utilizando los métodos que llamas en la clase File. Si intentas operar en un archivo que no existe o al que no tienes acceso, **algunos métodos de File lanzarán una excepción**. Otros métodos **devolverán false si el archivo no existe o la operación no se puede realizar**.

### **4. Trabajando con un Objeto File**

La clase File contiene **numerosos métodos útiles** para interactuar con archivos y directorios en el sistema de archivos. En la siguiente tabla se muestran los más importantes, por su uso:

### **5. Métodos más importantes de java.io.File**

|  |  |
| --- | --- |
| **Nombre del Método** | **Descripción** |
| boolean delete() | Borra el archivo o directorio y devuelve true sólo si la operación se completó con éxito. Si esta instancia es un directorio, el directorio debe estar vacío para poder eliminarse. |
| boolean exists() | Comprueba si un archivo existe |
| String getAbsolutePath() | Obtiene el nombre absoluto del archivo o directorio en el sistema de archivos |
| String getName() | Obtiene el nombre del archivo o directorio |
| String getParent() | Obtiene el directorio principal en el que se encuentra la ruta, o null si no hay ninguno |
| boolean isDirectory() | Comprueba si una referencia File es un directorio en el sistema de archivos |
| boolean isFile() | Comprueba si una referencia File es un archivo en el sistema de archivos |
| long lastModified() | Devuelve el número de milisegundos desde la época (número de milisegundos desde las 12 a.m. UTC del 1 de enero de 1970) en que se modificó el archivo por última vez |
| long length() | Obtiene el número de bytes en el archivo |
| File[] listFiles() | Obtiene una **lista de archivos** dentro de un directorio |
| boolean mkdir() | Crea el directorio especificado en la ruta |
| boolean mkdirs() | Crea el directorio especificado en la ruta, incluyendo cualquier directorio principal inexistente |
| boolean renameTo(File dest) | Cambia el nombre del archivo o directorio denotado por esta ruta a dest y devuelve true sólo si la operación tuvo éxito. |

Prueba el siempre útil programa de ejemplo de muestra que muestra información sobre un archivo o directorio, como si existe, qué archivos contiene y más:

import java.io.\*;  
import static java.lang.System.out;  
  
public class InfoFile {  
 public static void main(String args[]) throws IOException {  
 out.print("Raíz del sistema de ficheros");  
 for (File raiz: File.listRoots()) {  
 out.format("%s ", raiz);  
 }  
 out.println();  
 for (String nome : args) {  
 out.format("%n------%new File(%s)%n", nome);  
 File f = new File(nome);  
 out.format("toString(): %s%n", f);  
 out.format("exists(): %b%n", f.exists());  
 out.format("lastModified(): %tc%n", f.lastModified());  
 out.format("isFile(): %b%n", f.isFile());  
 out.format("isDirectory(): %b%n", f.isDirectory());  
 out.format("isHidden(): %b%n", f.isHidden());  
 out.format("canRead(): %b%n", f.canRead());  
 out.format("canWrite(): %b%n", f.canWrite());  
 out.format("canExecute(): %b%n", f.canExecute());  
 out.format("isAbsolute(): %b%n", f.isAbsolute());  
 out.format("length(): %d%n", f.length());  
 out.format("getName(): %s%n", f.getName());  
 out.format("getPath(): %s%n", f.getPath());  
 out.format("getAbsolutePath(): %s%n", f.getAbsolutePath());  
 out.format("getCanonicalPath(): %s%n", f.getCanonicalPath());  
 out.format("getParent(): %s%n", f.getParent());  
 out.format("toURI: %s%n", f.toURI());  
 }  
 }  
}

El siguiente es un programa de ejemplo de muestra que, dado una ruta a un archivo, muestra **información sobre el archivo o directorio**, si existe, qué archivos contiene y más:

var arquivo = new File("c:\\home\\otto\\noHayCole.txt");  
System.out.println("Archivo existe: " + arquivo.exists());  
if (arquivo.exists()) {  
  
 System.out.println("Ruta absoluta: " + arquivo.getAbsolutePath());  
  
 System.out.println("Es un directorio: " + arquivo.isDirectory());  
 System.out.println("Ruta padre: " + arquivo.getParent());  
  
 if (arquivo.isFile()) {  
 System.out.println("Tamaño: " + arquivo.length());  
 System.out.println("Última modificación: " + arquivo.lastModified());  
  
 } else {  
 for (File subArquivo : arquivo.listFiles()) {  
 System.out.println(" " + subArquivo.getName());  
 }  
 }  
}

Si la ruta proporcionada no apuntara a un archivo, produciría la siguiente salida:

Archivo existe: false

Si la ruta proporcionada apuntara a un archivo válido, produciría algo similar a lo siguiente:

Archivo existe: true  
Ruta absoluta: c:\home\otto\noHayCole.txt  
Es un directorio: false  
Ruta padre: c:\home\otto  
Tamaño: 14883  
Última modificación: 1806860000003

Finalmente, si la ruta proporcionada apuntara a un directorio válido, como c:\home, produciría algo similar a lo siguiente:

Archivo existe: true  
Ruta absoluta: c:\home  
Es un directorio: true  
Ruta padre: c:\  
asisoy.txt  
noHayCole.txt  
zalandomami.txt

En estos ejemplos, ves que la salida de un programa basado en Entrada/Salida **depende por completo de los directorios y archivos disponibles en tiempo de ejecución en el sistema de archivos subyacente**.

Directorio o archivo

Ojo, /home/otto/noHayCole.txt **podría ser un archivo o un directorio, incluso si tiene una extensión de archivo**. ¡No asumas que es uno u otro a menos que lo puedas comprobar! (por ejemplo, .git)

### **Ejercicios**

Ejercicio 1. Creación y lectura de archivos con File

*Debes* ***trabajar únicamente con métodos de la clase File****.*

Realiza los siguientes pasos:

1. **Crea un archivo de texto** llamado prueba.txt en el directorio actual de tu proyecto, sólo si no existe.
2. **Escribe un programa** que cree un objeto File para el archivo prueba.txt y **compruebe si el archivo existe**.
3. **Si el archivo existe**, muestra la **ruta absoluta**, **nombre del archivo**, **tamaño**, **última modificación** y **si es un directorio**.
4. **Si el archivo no existe**, muestra un mensaje que lo indique y crea uno temporal.

Ejercicio 2. Mostrar el contenido de un directorio

*Debes* ***trabajar únicamente con métodos de la clase File****.*

El programa abre una ventana para la selección de un directorio (hazlo también desde teclado si recoge un parámetro) y usando el **método listFiles()** de la clase File, **muestra el contenido de ese directorio**, indicando el tamaño de los archivos y si es un directorio o no. Además, muestra el tamaño total de los archivos y directorios.

Muestra en una ventana emergente el resultado y por consola.

A continuación puedes ver algunas soluciones parciales del ejercicio 2. Completa el ejercicio de acuerdo a las indicaciones.

#### **Solución parcial con list()**

import java.io.File;  
  
public class ListFiles {  
 public static void main(String[] args) {  
 File directorio = new File("C:\\Users\\Pepinho\\Documents\\GitHub\\dam2\\");  
 File[] archivos = directorio.listFiles();  
 for (File archivo : archivos) {  
 System.out.println(archivo.getName() + " " + archivo.length() + " " + (archivo.isDirectory() ? "Directorio" : "Archivo"));  
 }  
 }  
}

#### 

#### **Solución parcial con JFileChooser**

import javax.swing.JFileChooser;  
import java.io.File;  
  
public class ListFiles {  
 public static void main(String[] args) {  
 JFileChooser fileChooser = new JFileChooser();  
 fileChooser.setFileSelectionMode(JFileChooser.DIRECTORIES\_ONLY);  
 fileChooser.showOpenDialog(null);  
 File directorio = fileChooser.getSelectedFile();  
 File[] archivos = directorio.listFiles();  
 for (File archivo : archivos) {  
 System.out.println(archivo.getName() + " " + archivo.length() + " " + (archivo.isDirectory() ? "Directorio" : "Archivo"));  
 }  
 }  
}

**Solución completa con JFileChooser**

import javax.swing.JFileChooser;  
import java.io.File;  
  
public class ListFiles {  
 public static void main(String[] args) {  
 JFileChooser fileChooser = new JFileChooser();  
 fileChooser.setFileSelectionMode(JFileChooser.DIRECTORIES\_ONLY);  
 fileChooser.showOpenDialog(null);  
 File directorio = fileChooser.getSelectedFile();  
 File[] archivos = directorio.listFiles();  
 long total = 0;  
 for (File archivo : archivos) {  
 System.out.println(archivo.getName() + " " + archivo.length() + " " + (archivo.isDirectory() ? "Directorio" : "Archivo"));  
 total += archivo.length();  
 }  
 System.out.println("Tamaño total: " + total);  
 }  
}

El siguiente ejemplo muestra cómo **mostrar el contenido de un directorio**, haciendo uso de la clase que veremos BufferesReader (no Scanner) para la lectura de teclado:

// Programa Java que muestra todo el contenido de un directorio  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
// Mostrando el contenido de un directorio  
class Contents {  
 public static void main(String[] args)  
 throws IOException  
 {  
 // Introducimos la ruta y el nombre del directorio por teclado:  
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
  
 System.out.println("Introduce la ruta:");  
 String dirpath = br.readLine();  
 System.out.println("Introduce el nombre del directorio:");  
 String dname = br.readLine();  
  
 // creamos un objeto File a partir de la ruta y el nombre del directorio  
 File f = new File(dirpath, dname);  
  
 // si el directorio existe, mostramos su contenido  
 if (f.exists()) {  
 // obtenemos el contenido en un arr[]  
 // el array arr[i] representa el nombre cada archivo o directorio  
 String arr[] = f.list();  
  
 // Número de entradas en el directorio  
 int n = arr.length;  
  
 // mostramos cada una de las entradas.  
 for (int i = 0; i < n; i++) {  
 System.out.println(arr[i]);  
 // Creamos un objeto File para cada entrada y   
 // comprobamos si es un archivo o un directorio.  
 File f1 = new File(arr[i]);  
 if (f1.isFile())  
 System.out.println(": es un archivo");  
 if (f1.isDirectory())  
 System.out.println(": es un directorio");  
 }  
 System.out.println("El directorio no tiene entradas " + n);  
 }  
 else  
 System.out.println("Directorio no encontrado");  
 }  
}

Ejercicio 3. Gestor de archivos y directorios

Como en todos los ejercicios anteriores, debes **trabajar únicamente con métodos de la clase File**.

Escribe un programa en Java que funcione como un **gestor básico de archivos y directorios**. El programa debe permitir al usuario realizar las siguientes operaciones:

1. **Crear** un directorio, empleando la clase JFileChooser para seleccionar la ruta donde se creará.
2. **Listar** todos los archivos y subdirectorios de un directorio **de forma recursiva**.
3. **Eliminar** un archivo o directorio. Si es un directorio, eliminar todo su contenido de forma recursiva.
4. **Mover o renombrar** archivos y directorios.

El programa debe ofrecer un menú para que el usuario elija la operación que desea realizar. La selección de directorios o archivos debe realizarse con la clase JFileChooser.

### **6. Nuevas características del paquete java.nio.file**

Aunque la clase java.io.File es útil para muchas operaciones de E/S de archivos, **Java SE 7** **introdujo una nueva API de E/S de archivos en el paquete java.nio.file que proporciona una funcionalidad más rica y más eficiente para trabajar con archivos y directorios**. Este modo de hacerlo lo veremos en el siguiente apartado.

Antes del lanzamiento de **Java SE 7**, la clase java.io.File era el mecanismo utilizado para la E/S de archivos, pero presentaba varios inconvenientes:

* **Muchos métodos no lanzaban excepciones al fallar**, por lo que era imposible obtener un mensaje de error útil. Por ejemplo, si fallaba la eliminación de un archivo, el programa recibía un “fallo al eliminar”, pero no sabía si era porque el archivo no existía, el usuario no tenía permisos, o había algún otro problema.
* El método **rename no funcionaba de manera consistente en diferentes plataformas**.
* **No había un soporte real para enlaces simbólicos**.
* Se **requería más soporte para metadatos**, como permisos de archivos, propietario del archivo y otros atributos de seguridad.
* El **acceso a los metadatos de los archivos era ineficiente**.
* **Muchos de los métodos no escalaban bien**. Solicitar un listado de directorios grandes en un servidor podía causar bloqueos. Los directorios grandes también podían generar problemas de recursos de memoria, lo que resultaba en una denegación de servicio.
* **No era posible escribir código fiable que pudiera recorrer un árbol de archivos recursivamente** y responder adecuadamente si había enlaces simbólicos circulares.

Aun así, **existe mucho código que usa java.io.File y sigue siendo útil para muchas situaciones**. Aunque lo veremos al detalle, si quisieras aprovechar la funcionalidad de java.nio.file.Path con el menor impacto posible en tu código muestro ejemplos de ello.

#### **Conversión entre java.io.File y java.nio.file.Path**

La **clase java.io.File proporciona el método toPath, que convierte una instancia de estilo antiguo en una instancia java.nio.file.Path**:

Path entrada = file.toPath();

De esta forma, puedes aprovechar el conjunto de características que ofrece la clase Path.

Por ejemplo, si tuvieras algún código que eliminara un archivo:

file.delete();

Podrías modificar este código para usar el método Files.delete, de la siguiente manera:

Path fp = file.toPath();  
Files.delete(fp);

A la inversa, **el método Path.toFile construye un objeto java.io.File para un objeto Path**.

#### **Mapeo de la Funcionalidad de java.io.File a java.nio.file**

Dado que la implementación de la E/S de archivos en Java ha sido completamente re-arquitectada en la versión **Java SE 7**, **no puedes intercambiar un método por otro directamente**. Si deseas usar la rica funcionalidad que ofrece el paquete java.nio.file, la solución más sencilla es **usar el método File.toPath**.

**No hay una correspondencia uno a uno entre las dos APIs**, pero la siguiente tabla da una idea general de qué funcionalidad en la API java.io.File corresponde a la funcionalidad en la API java.nio.file, y te indica dónde puedes obtener más información.

|  |  |  |
| --- | --- | --- |
| **Funcionalidad de java.io.File** | **Funcionalidad de java.nio.file** | **Uso** |
| java.io.File | java.nio.file.Path | Clase principal de gestión de archivos. |
| java.io.RandomAccessFile | SeekableByteChannel | Archivos de Acceso Aleatorio |
| File.canRead, canWrite, canExecute | Files.isReadable, Files.isWritable, Files.isExecutable | Verificación de archivo o directorio. |
| File.isDirectory(), File.isFile(), File.length() | Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...), Files.size(Path) | Gestión de Metadatos de archivo/directorio. |
| File.lastModified(), File.setLastModified(long) | Files.getLastModifiedTime(Path, LinkOption...), Files.setLastModifiedTime(Path, FileTime) | Gestión de Metadatos de fecha modificación. |
| Métodos que establecen varios atributos (setExecutable, setReadable, setReadOnly, setWritable) | Files.setAttribute(Path, String, Object, LinkOption...) | Gestión de Metadatos de atributos de archivo. |
| new File(parent, "newfile") | parent.resolve("newfile") | Operaciones con archivos |
| File.renameTo | Files.move | Mover un Archivo o Directorio |
| File.delete | Files.delete | Eliminar un Archivo o Directorio |
| File.createNewFile | Files.createFile | Crear Archivos |
| File.deleteOnExit | Opción DELETE\_ON\_CLOSE especificada en createFile | Borrado de archivos al salir. |
| File.createTempFile | Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path, String, String, FileAttributes<?>) | Crear Archivos temporales. |
| File.exists | Files.exists, Files.notExists | Verificar la Existencia de un Archivo o Directorio |
| File.compareTo, equals | Path.compareTo, equals | Comparar dos archivos/paths |
| File.getAbsolutePath, getAbsoluteFile | Path.toAbsolutePath | Obtención de la ruta absoluta. |
| File.getCanonicalPath, getCanonicalFile | Path.toRealPath o normalize | Convertir un Path (toRealPath), Eliminar Redundancias en un Path (normalize) |
| File.toURI | Path.toURI | Convertir un path en una URL. |
| File.isHidden | Files.isHidden | Saber si está oculto. |
| File.list, listFiles | Path.newDirectoryStream | Listar el Contenido de un Directorio |
| File.mkdir, mkdirs | Files.createDirectory | Crear directorio/s |
| File.listRoots | FileSystem.getRootDirectories | Listar los Directorios Raíz del Sistema de Archivos |
| File.getTotalSpace, getFreeSpace, getUsableSpace | FileStore.getTotalSpace, getUnallocatedSpace, getUsableSpace, getTotalSpace | Atributos del Almacenamiento de Archivos |

### **7. La clase java.io.RandomAccessFile**

La **clase** [**RandomAccessFile**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/RandomAccessFile.html) permite **acceso no secuencial, o aleatorio, al contenido del archivo**.

Permite **leer y escribir (implementa las interfaces DataInput y DataOutput) en archivos de acceso aleatorio**. En el constructor **se especifica el modo de apertura, lectura o escritura**:

new RandomAccessFile("proba.txt", "r"); // Solo lectura  
new RandomAccessFile("proba.txt", "rw"); // Lectura y escritura  
new RandomAccessFile("proba.txt", "rwd"); // Lectura y escritura, sincronizado

* Emplea la notación de **puntero a archivo para especificar la posición actual en el archivo**.
* Al crearlo apunta al **principio del archivo**, la **posición 0**.

Las **sucesivas llamadas** a **read** o **write** **modifican la posición del punter**o el número de bytes leídos o escritos, respectivamente.

Dispone de 3 métodos para modificar la posición del puntero:

* int skipBytes(int n): mueve el puntero hacia delante n bytes.
* void seek(long): sitúa el puntero justo antes del byte especificado.
* long getFilePointer(): devuelve la posición actual del puntero a archico.

Definición de la clase RandomAccessFile:

public class RandomAccessFile   
 extends Object implements DataOutput, DataInput, Closeable

Las instancias de esta clase soportan tanto la **lectura como la escritura en un archivo de acceso aleatorio**. Un archivo de acceso aleatorio se comporta como un **gran array de bytes almacenado en el sistema de archivos**. Existe un tipo de cursor, o índice en el array implícito, llamado **puntero de archivo**; las operaciones de entrada leen bytes comenzando en el puntero de archivo y avanzan el puntero más allá de los bytes leídos.

Si el archivo de acceso aleatorio se crea en modo de lectura/escritura, entonces las operaciones de salida también están disponibles; las operaciones de salida escriben bytes comenzando en el puntero de archivo y avanzan el puntero más allá de los bytes escritos. Las operaciones de salida que escriben más allá del final actual del array implícito causan que el array se extienda.

El **puntero de archivo se puede leer mediante el método getFilePointer y establecer mediante el método seek**.

Para todas las rutinas de lectura en esta clase que, **si se alcanza el final del archivo antes de que se haya leído el número deseado de bytes**, se lanza una excepción **EOFException** (que es un tipo de **IOException**).

**Si no se puede leer ningún byte** por alguna razón que no sea el final del archivo, se lanza una **IOException** distinta a **EOFException**. En particular, puede lanzarse una **IOException** si el flujo ha sido cerrado.

Ejemplo de uso de la clase RandomAccessFile:

import java.io.\*;  
  
public class RandomAccessFileDemo {  
 public static void main(String[] args) {  
 try {  
 RandomAccessFile raf = new RandomAccessFile("proba.txt", "rw");  
 raf.writeUTF("Hola, mundo!");  
 raf.seek(0);  
 System.out.println(raf.readUTF());  
 raf.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

#### **Escritura con RandomAccessFile**

Ahora veremos **cómo escribir y editar dentro de un archivo existente**, en lugar de solo escribir en un archivo completamente nuevo o agregar a uno existente. Simplemente: necesitamos acceso aleatorio.

**RandomAccessFile** nos **permite escribir en una posición específica del archivo**, **dado el desplazamiento (offset) desde el principio del archivo en bytes**.

Este código escribe un valor entero con un desplazamiento dado desde el principio del archivo:

private void writeToPosition(String filename, int data, long position)   
 throws IOException {  
 RandomAccessFile writer = new RandomAccessFile(filename, "rw");  
 writer.seek(position);  
 writer.writeInt(data);  
 writer.close();  
}

Si queremos leer el entero almacenado en una ubicación específica, podemos usar este método:

private int readFromPosition(String filename, long position)   
 throws IOException {  
 int result = 0;  
 RandomAccessFile reader = new RandomAccessFile(filename, "r");  
 reader.seek(position);  
 result = reader.readInt();  
 reader.close();  
 return result;  
}

Para probar nuestras funciones, escribamos un entero, lo editemos, y finalmente lo leamos:

@Test  
public void whenWritingToSpecificPositionInFile\_thenCorrect()   
 throws IOException {  
 int data1 = 2014;  
 int data2 = 1500;  
   
 writeToPosition(fileName, data1, 4);  
 assertEquals(data1, readFromPosition(fileName, 4));  
   
 writeToPosition(fileName2, data2, 4);  
 assertEquals(data2, readFromPosition(fileName, 4));  
}

### **Ejercicios**

Ejercicio 4. Escritura y lectura de archivos con RandomAccessFile

Escribe un programa que **escriba y lea datos en un archivo** usando la clase RandomAccessFile.

1. **Crea un archivo de texto** llamado prueba.txt en el directorio actual de tu proyecto, sólo si no existe.
2. **Escribe un programa** que cree un objeto RandomAccessFile para el archivo prueba.txt y **escriba un mensaje**.
3. **Lee el mensaje** y **muéstralo por consola**.

Ejercicio 5. Escritura y lectura de archivos con RandomAccessFile

Escribe un programa que utilice la clase RandomAccessFile para escribir en un archivo los números del 1 al 10 y luego los lea desde el archivo. Muestra los números leídos en la consola.

#### **Solución al ejercicio 5**

import java.io.IOException;  
import java.io.RandomAccessFile;  
  
public class RandomAccessFileDemo {  
 public static void main(String[] args) {  
 try {  
 RandomAccessFile raf = new RandomAccessFile("prueba.txt", "rw");  
 for (int i = 1; i <= 10; i++) {  
 raf.writeInt(i);  
 }  
 raf.seek(0);  
 for (int i = 1; i <= 10; i++) {  
 System.out.println(raf.readInt());  
 }  
 raf.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
   
 }  
 }  
}

Ejercicio 6. Modificación de Contenido en un Archivo Binario con `RandomAccessFile`

Escribe un programa en Java que haga lo siguiente:

* Escriba 10 enteros en un archivo llamado “datos.bin”.
* Permita al usuario modificar el tercer número almacenado en el archivo por otro número.
* Muestra los números antes y después de la modificación en la consola.

#### **Solución al ejercicio 6**

import java.io.RandomAccessFile;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class Ejercicio3 {  
 public static void main(String[] args) {  
 try (RandomAccessFile raf = new RandomAccessFile("datos.bin", "rw")) {  
 // Escribir 10 enteros en el archivo  
 for (int i = 1; i <= 10; i++) {  
 raf.writeInt(i);  
 }  
  
 // Leer los números antes de la modificación  
 System.out.println("Números antes de la modificación:");  
 raf.seek(0);  
 for (int i = 0; i < 10; i++) {  
 System.out.println(raf.readInt());  
 }  
  
 // Solicitar al usuario un nuevo número para el tercer número  
 Scanner sc = new Scanner(System.in);  
 System.out.print("Ingrese un nuevo número para reemplazar el tercer número: ");  
 int nuevoNumero = sc.nextInt();  
  
 // Modificar el tercer número (posición 2 en base 0, cada entero ocupa 4 bytes)  
 raf.seek(2 \* 4);  
 raf.writeInt(nuevoNumero);  
  
 // Leer los números después de la modificación  
 System.out.println("Números después de la modificación:");  
 raf.seek(0);  
 for (int i = 0; i < 10; i++) {  
 System.out.println(raf.readInt());  
 }  
  
 } catch (IOException e) {  
 System.out.println("Ocurrió un error de entrada/salida.");  
 e.printStackTrace();  
 }  
 }  
}

## **01.02 LA CLASE RANDOMACCESSFILE**

* [La Clase RandomAccessFile en Java](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#la-clase-randomaccessfile-en-java)
  + [1. Creación de un RandomAccessFile](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#1-creaci%C3%B3n-de-un-randomaccessfile)
  + [2. Modos de Acceso](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#2-modos-de-acceso)
  + [3. Situar el puntero: seek()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#3-situar-el-puntero-seek)
  + [4. Posición actual del puntero: getFilePointer()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#4-posici%C3%B3n-actual-del-puntero-getfilepointer)
  + [5. Lectura de un Byte desde: read()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#5-lectura-de-un-byte-desde-read)
  + [6. Lectura de un array de bytes: read(byte[])](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#6-lectura-de-un-array-de-bytes-readbyte)
  + [7. Escritura de un byte: write()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#7-escritura-de-un-byte-write)
  + [8. Escritura de un array de bytes: write(byte[])](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#8-escritura-de-un-array-de-bytes-writebyte)
  + [9. Cierre del archivo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#9-cierre-del-archivo)
  + [Ejemplo completo del uso de RandomAccessFile](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0102randomaccessfile/#ejemplo-completo-del-uso-de-randomaccessfile)

### **La Clase RandomAccessFile en Java**

La clase RandomAccessFile de Java en la API de Java IO te **permite navegar por un archivo y leer o escribir en él según sea necesario**. También puedes **reemplazar partes existentes de un archivo**. Esto no es posible con FileInputStream o FileOutputStream, que veremos en el apartado de flujos de E/S.

### **1. Creación de un RandomAccessFile**

Antes de poder trabajar con la clase RandomAccessFile, debes crear una instancia de esa clase:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");

Nota el **segundo parámetro del constructor**, "rw",**es el modo en el que quieres abrir el archivo**. "rw" significa modo de lectura/escritura..

### **2. Modos de Acceso**

La clase RandomAccessFile de Java soporta los siguientes modos de acceso:

|  |  |
| --- | --- |
| **Modo** | **Descripción** |
| r | Modo de **lectura**. Llamar a los métodos de escritura lanzará en una IOException. |
| rw | Modo de **lectura y escritura**. |
| rwd | Modo de **lectura y escritura** - sincrónicamente. Todas las actualizaciones al contenido del archivo se escriben en el disco de manera sincrónica. |
| rws | Modo de lectura y escritura - sincrónicamente. Todas las actualizaciones al **contenido del archivo o metadatos** se escriben en el disco de manera sincrónica. |

### **3. Situar el puntero: seek()**

Para leer o escribir en una ubicación específica en un RandomAccessFile, primero debes **situar el puntero del archivo (también llamado seek) en la posición de lectura o escritura**. Esto se hace utilizando el método seek(). Por ejemplo:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");  
file.seek(100);

### **4. Posición actual del puntero: getFilePointer()**

Puedes obtener la **posición actual de un RandomAccessFile usando su método getFilePointer()**. La posición actual es el índice (desplazamiento) del **byte** en el que el RandomAccessFile está actualmente situado:

long posicion = file.getFilePointer();

### **5. Lectura de un Byte desde: read()**

La **lectura** un byte desde un RandomAccessFile **se realiza usando su método read()**:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");  
int miByte = file.read();

El método **read() lee el byte ubicado en la posición del archivo señalada por el puntero** en la instancia de RandomAccessFile.

Avance del puntero

Un detalle que el javadoc olvida mencionar: **el método read() incrementa el puntero del archivo para que apunte al siguiente byte** después del que acaba de ser leído. Esto significa **se puede seguir llamando a read() sin tener que mover manualmente el puntero del archivo**.

### **6. Lectura de un array de bytes: read(byte[])**

También es posible **leer un array de bytes** con un RandomAccessFile:

RandomAccessFile randomAccessFile = new RandomAccessFile("programas/datos.txt", "r");  
  
byte[] dest = new byte[1024]; // Array de bytes donde se almacenarán los datos leídos, llamado buffer.  
int offset = 0;  
int length = 1024;  
int bytesLeidos = randomAccessFile.read(dest, offset, length);

Este ejemplo lee una secuencia de bytes en el array de bytes dest pasado como parámetro al método read(). **El método read() comenzará a leer en el archivo desde la posición actual del puntero del archivo** en el RandomAccessFile. El método read() **comenzará a leer datos en el array de bytes a partir de la posición proporcionada por el parámetro offset**, y como máximo el número de bytes proporcionado por el parámetro length.

Este método **devuelve el número real de bytes leídos**.

### **7. Escritura de un byte: write()**

Puedes escribir un byte en un RandomAccessFile **usando su método write()**, el cual toma un entero como parámetro. El byte se escribirá en la posición actual del puntero del archivo en el RandomAccessFile. **El byte anterior en esa posición será sobrescrito**:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");  
file.write(67); // Código ASCII para 'C'

Recuerda, **llamar a este método write() avanzará la posición del archivo en 1 byte**, al igual que sucede con el método read().

### **8. Escritura de un array de bytes: write(byte[])**

Escribir en un RandomAccessFile se puede hacer usando **uno de sus muchos métodos write()**:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");  
  
byte[] bytes = "Hello World".getBytes("UTF-8");  
file.write(bytes);

Este ejemplo **escribe el array de bytes en la posición actual** del puntero del archivo en el objeto RandomAccessFile. Cualquier **byte que esté en esa posición será sobrescrito** con los nuevos bytes.

Al igual que con el método read(), el método write() **avanza el puntero del archivo después de ser llamado**. De esta manera no tienes que mover constantemente el puntero para escribir datos en una nueva ubicación en el archivo.

También puedes escribir partes de un array de bytes en un RandomAccessFile, en lugar de todo el array:

RandomAccessFile file = new RandomAccessFile("c:\\data\\holamundo.kt", "rw");  
  
byte[] bytes = "Hello World".getBytes("UTF-8");  
file.write(bytes, 2, 5);

Este ejemplo escribe desde el **desplazamiento** (*offset*) **2 del array de bytes y 5 bytes hacia adelante**, **longitud** (*length*).

### **9. Cierre del archivo**

El RandomAccessFile tiene un método **close() que debe ser llamado cuando hayas terminado de usar la instancia de RandomAccessFile**:

RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw");  
file.close();

También puedes **cerrar un RandomAccessFile automáticamente si usas la sentencia *try-with-resources* de Java**:

try (RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.kt", "rw")) {  
  
 // lectura o escritura en el RandomAccessFile  
  
}

Una vez que la ejecución del programa salga del bloque **try-with-resources, el objeto RandomAccessFile se cerrará automáticamente**, incluso si se lanza una IOException desde dentro del bloque try-with-resources.

### **Ejemplo completo del uso de RandomAccessFile**

En el siguiente ejemplo escribimos una lista de estudiantes pedidos por teclado, guardando el número de estudiantes y el nombre en el mismo archivo. Para la lectura solicitamos el número del estudiante a leer:

import java.io.IOException;  
import java.io.RandomAccessFile;  
import java.util.Scanner;  
  
public class RegistroEstudiantes {  
  
 public static void main(String[] args) throws IOException { // En realidad es mala opción lanzar la excepción, pero es para simplificar el ejemplo  
  
 try (RandomAccessFile file = new RandomAccessFile("E:\\programas\\estudiantes.txt", "rw")) {  
  
 Scanner scanner = new Scanner(System.in);  
  
 System.out.println("Introduce el número de estudiantes: ");  
 int numEstudiantes = scanner.nextInt();  
 file.writeInt(numEstudiantes);  
  
 for (int i = 0; i < numEstudiantes; i++) {  
 System.out.println("Introduce el nombre del estudiante " + (i + 1) + ": ");  
 String nombre = scanner.next();  
 file.writeUTF(nombre);  
 }  
  
 System.out.println("Introduce el número del estudiante a leer: ");  
 int numEstudiante = scanner.nextInt();  
  
 file.seek(0);  
 int numEstudiantesGuardados = file.readInt();  
  
 if (numEstudiante > numEstudiantesGuardados) {  
 System.out.println("No hay tantos estudiantes guardados.");  
 } else {  
 file.seek(4); // Saltamos el número de estudiantes  
 for (int i = 0; i < numEstudiante - 1; i++) {  
 file.readUTF();  
 }  
 System.out.println("El estudiante " + numEstudiante + " es: " + file.readUTF());  
 }  
 }  
 }  
}

## 01.03 FLUJOS DE E/S

* [1. Introducción a los flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#1-introducci%C3%B3n-a-los-flujos-de-es)
  + [Flujo de entrada](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#flujo-de-entrada)
  + [Flujo de salida](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#flujo-de-salida)
    - [Tipos de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#tipos-de-datos)
* [2. Fundamentos de los flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#2-fundamentos-de-los-flujos-de-es)
* [3. Nomenclatura de los flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#3-nomenclatura-de-los-flujos-de-es)
* [4. Flujos de bytes vs. flujos de caracteres](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#4-flujos-de-bytes-vs-flujos-de-caracteres)
  + [4.1. Flujos de bytes (Byte Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#41-flujos-de-bytes-byte-streams)
  + [4.2. Flujos de caracteres (Character Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#42-flujos-de-caracteres-character-streams)
* [5. Flujos de entrada (Input Streams) vs. flujos de salida (Output Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0103javaiostreams/#5-flujos-de-entrada-input-streams-vs-flujos-de-salida-output-streams)

### 1. Introducción a los flujos de E/S

Ahora que hemos cubierto los conceptos básicos de la clase File, pasemos a los flujos (streams) de E/S, que son mucho más interesantes, pues **no sólo pueden emplearse para archivos**.

Un **flujo de E/S representa una fuente de entrada o un destino de salida**. Un flujo puede representar muchos tipos diferentes de fuentes y destinos, incluidos **archivos en disco, dispositivos, otros programas, String o arrays de memoria**.

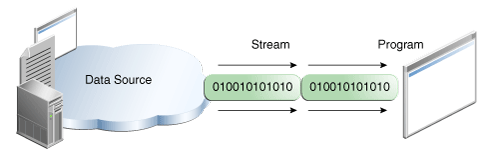
En esta sección, veremos **cómo usar los flujos de E/S para leer y escribir datos**. La “E/S” se refiere a la naturaleza de cómo se accede a los datos, ya sea **leyendo los datos desde un recurso (entrada) o escribiendo los datos en un recurso (salida)**.

Flujos de E/S en Java

En Java, los flujos de E/S se encuentran en el paquete java.io. Aunque Java 9 introdujo un nuevo paquete java.nio.file para operaciones de E/S más avanzadas, java.io sigue siendo ampliamente utilizado y es importante comprenderlo.

Los flujos **admiten muchos tipos diferentes de datos, incluidos bytes simples, tipos de datos primitivos, caracteres localizados y objetos**. Algunos flujos simplemente transmiten datos; otros manipulan y transforman los datos de formas útiles.

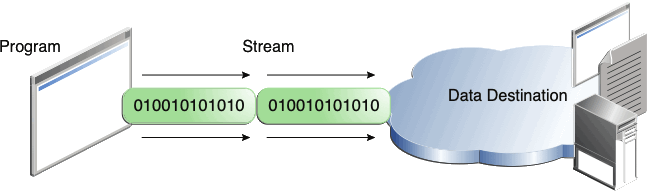
#### Flujo de entrada



Representan una **fuente de entrada**. Pueden proceder de diferentes **tipos de fuentes**:

* **Archivos** de disco ([FileReader](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileReader.html), [FileInputStream](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileInputStream.html), [FIleImageInputStream](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/imageio/stream/FileImageInputStream.html),…).
* **Dispositivos**: teclado ([System.in](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/System.html#in)),…
* Otros **programas**.
* **Arrays** de memoria (StringBufferInputStream (desaprobado, por [StringReader](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/StringReader.html)),…

#### Flujo de salida



* Representan un **destino de salida**.
* Puede representar diferentes **tipos de destinos**:
  + **Archivos** de disco: [FileWriter](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileWriter.html), [FileOutputStream](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileOutputStream.html), [FileImageOutputStream](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/imageio/stream/FileImageOutputStream.html),…
  + **Dispositivos**: pantalla ([System.out](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/System.html#out)),…
  + Otros **programas**.
  + **Arrays** de memoria: [ByteArrayOutputStream](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/ByteArrayOutputStream.html),…

##### Tipos de datos

Ambos tipos de flujo pueden representar **diferentes tipos de datos**:

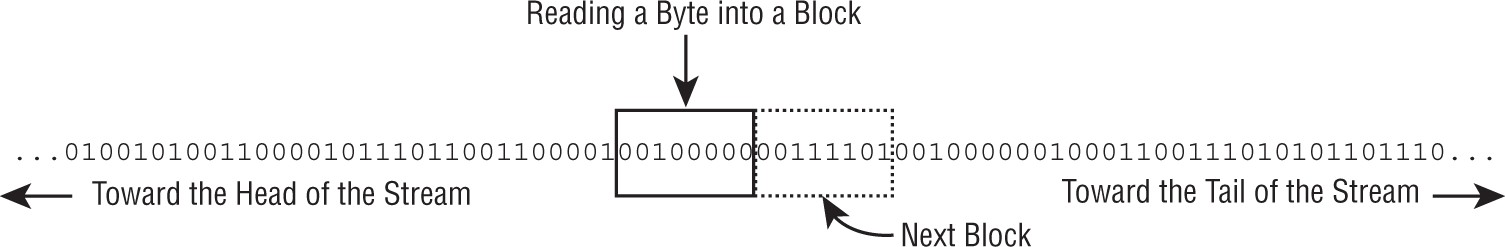
* **Bytes** simples. (FileInputStream, FileOutputStream,…).
* Tipos de **datos primitivos** (DataInputStream,…).
* **Caracteres** (FileReader, FileWriter,…).
* **Objetos** (ObjectInputStream,…).
* Algunos flujos **simplemente pasan datos, otros manipulan y transforman los datos**.

### 2. Fundamentos de los flujos de E/S

El contenido de un **archivo, una página Web, el teclado, etc. se puede leer o escribir a través de un *flujo***, que es una lista de elementos de **datos presentados secuencialmente**. Deberías pensar en los flujos conceptualmente como un “flujo de agua” largo y casi interminable con datos que se presentan uno a uno, como una “ola” a la vez.

En general, **el flujo es tan grande que una vez que comenzamos a leerlo, no tenemos idea de dónde comienza o termina**. Sólo tenemos un **puntero a nuestra posición actual en el flujo y leemos datos bloque por bloque**.

**Cada tipo de flujo segmenta los datos en una “chorro” o “bloque” de una manera particular**. Por ejemplo, algunas clases de flujos leen o escriben datos como bytes individuales. Otras clases de flujos leen o escriben caracteres individuales o cadenas de caracteres. Además, **algunas clases de flujos leen o escriben grupos más grandes de bytes o caracteres a la vez, específicamente aquellas con la palabra “*Buffered*” en su nombre**.



Aunque los flujos se utilizan comúnmente con la E/S de archivos, **se utilizan de manera más general para manejar la lectura/escritura de cualquier fuente de datos de flujos**. Por ejemplo, podrías construir una aplicación Java que envíe datos a un sitio web utilizando un flujo de salida y lea el resultado a través de un flujo de entrada.

Entrada vs Salida

Es importante **distinguir entre entrada (InputStream/Reader) y salida (OutputStream/Writer)**. Es muy sencillo, pues siempre debe verse desde el punto de vista del programa: **entrada de datos al programa (lectura) y salida de datos desde el programa (escritura)**.

### 3. Nomenclatura de los flujos de E/S

La **API java.io proporciona numerosas clases para crear, acceder y manipular flujos**, tantas que tienden a abrumar a muchos desarrolladores de Java. ¡Mantén la calma! ;-) Revisaremos las principales diferencias entre cada clase de flujo y veremos cómo distinguirlas. A menudo **el nombre del flujo te proporciona suficiente información para comprender exactamente qué hace**.

El objetivo de este apartado es familiarizarte con la terminología común y las convenciones de nombres utilizadas con los flujos. No te preocupes si no reconoces los nombres de las clases de flujos particulares que se utilizan aquí; los veremos más adelante y con la práctica se entenderá mejor.

### 4. Flujos de bytes vs. flujos de caracteres

La API java.io define **dos conjuntos de clases de flujos para la lectura y escritura de flujos**: **flujos de bytes y flujos de caracteres**.

#### 4.1. Flujos de bytes (Byte Streams)

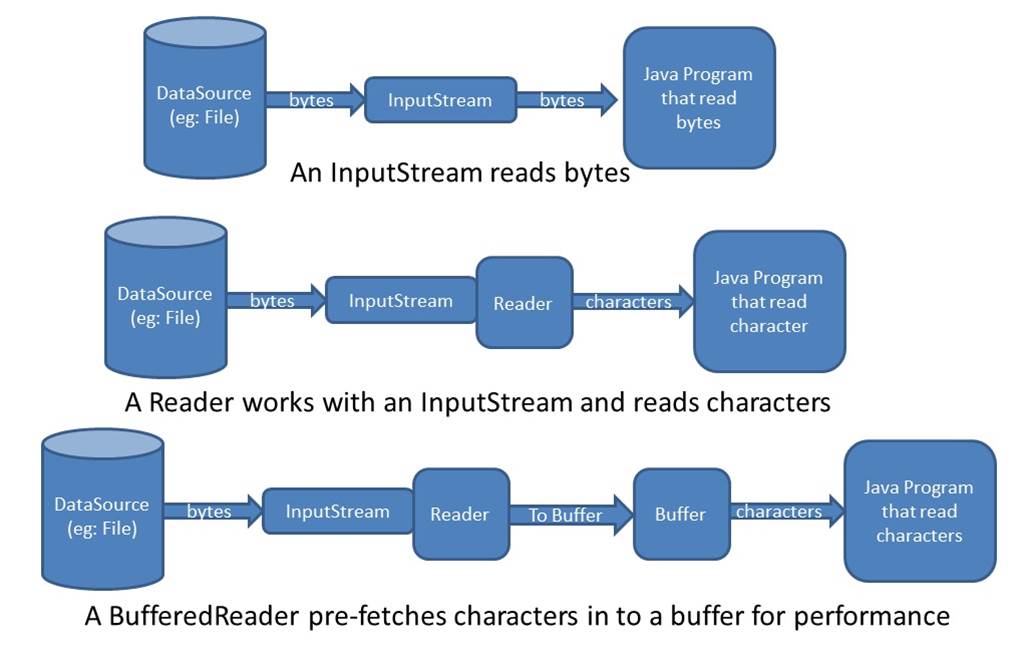
* **Los flujos de bytes leen/escriben datos binarios (0 y 1)** y tienen nombres de clase que **terminan en InputStream o OutputStream**.
* **Todas** las clases **descienden (heredan) de** [**InputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/InputStream.html) **y** [**OutputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/OutputStream.html).
* Hay muchas clases de flujos de bytes, como: [**FileInputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileInputStream.html) **y** [**FileOutputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileOutputStream.html). Todos los restantes flujos funcionan del mismo modo sólo difieren en la forma de construirlos.

Los programas utilizan **flujos de bytes para realizar la entrada y salida de bytes de 8 bits**. Todas las clases de flujos de bytes **heredan de** [**InputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/InputStream.html) **y** [**OutputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/OutputStream.html).

#### 4.2. Flujos de caracteres (Character Streams)

* Los flujos de caracteres **leen/escriben datos de texto** y tienen nombres de clase que **terminan en Reader o Writer**.
* Automáticamente, **transforma caracteres Unicode (formato de Java) al conjunto de caracteres local**.
* Todas las clases **descienden de Reader y Writer**.
* Hay muchas clases de flujos de carácter, como :[FileReader (usa internamente FileInputStream)](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileReader.html), [FileWriter (usa internamente **FileOutpuStream**)](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileWriter.html). Todos los restantes flujos funcionan de igual modo, sólo difieren en la forma de construirlos.

**Java almacena valores de caracteres utilizando convenciones Unicode.** La E/S de flujos de caracteres **traduce automáticamente este formato interno hacia y desde el conjunto de caracteres local**. En locales occidentales, como el juego de caracteres Latin-1 o Windows-1252, el conjunto de caracteres local es generalmente un superconjunto de ASCII de 8 bits. En locales asiáticos, el conjunto de caracteres local es un conjunto de caracteres de doble byte.



### 5. Flujos de entrada (Input Streams) vs. flujos de salida (Output Streams)

La mayoría de las clases de **flujos de entrada tienen una clase de flujo de salida correspondiente, y viceversa**. Por ejemplo, la clase FileOutputStream escribe datos que pueden ser leídos por un FileInputStream. Si comprendes las características de una clase de flujo de entrada o salida en particular, naturalmente sabrás qué hace su clase complementaria.

Por lo tanto, **la mayoría de las clases Reader tienen una clase Writer correspondiente**. Por ejemplo, la clase FileWriter escribe datos que pueden ser leídos por un FileReader. Aunque hay excepciones a esta regla:

La soledad de los flujos de salida `PrintWriter` y `PrintStream`

Debes saber que **[PrintWriter][printwriter] no tiene una clase ~~PrintReader~~ correspondiente**.

Del mismo modo, [**PrintStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/PrintStream.htm) **es una OutputStream que no tiene una clase InputStream correspondiente**. Tampoco tiene la palabra “Output” en su nombre.

El principal propósito de PrintWriter y PrintStream es **facilitar la escritura de datos formateados en un flujo**, como sucede con [**System.out**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/System.html#out) **y** [**System.err**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/System.html#out) **que son de tipo PrintStream**.

Hablaremos de estas clases más adelante.

## 01.04 FLUJOS DE BYTE

* [Flujos de bytes (Byte Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#flujos-de-bytes-byte-streams)
  + [Ejemplo: copia de archivos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#ejemplo-copia-de-archivos)
  + [Cierre de flujos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#cierre-de-flujos)
  + [1. InputStream](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#1-inputstream)
  + [2. OutputStream](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#2-outputstream)
  + [3. ObjectInputStream y ObjectOutputStream](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#3-objectinputstream-y-objectoutputstream)
  + [4. Lectura desde URL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#4-lectura-desde-url)
    - [URI/URL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#uriurl)
    - [URLConnection](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#urlconnection)
    - [HttpURLConnection](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0104bytestream/#httpurlconnection)

### Flujos de bytes (Byte Streams)

* **Los flujos de bytes leen/escriben datos binarios (0 y 1)** y tienen nombres de clase que terminan en InputStream o OutputStream.
* Leen en **bloques de bytes y no pueden manejar caracteres Unicode**.
* Todas las clases **descienden (heredan) de** [**InputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/InputStream.html) **y** [**OutputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/OutputStream.html).
* Hay **muchas clases de flujos de bytes, como: FileInputStream y FileOutputStream**. Todos los restantes flujos funcionan del mismo modo sólo difieren en la forma de construirlos.

Los programas utilizan **flujos de bytes para realizar la entrada y salida de bytes de 8 bits**. Todas las clases de flujos de bytes **heredan de** [**InputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/InputStream.html) **y** [**OutputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/OutputStream.html).

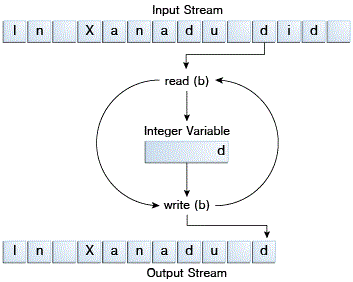
Veremos un **ejemplo** de cómo funcionan los **flujos de bytes con flujos de bytes de E/S de archivo, FileInputStream y FileOutputStream**. Otros tipos de flujos de bytes se utilizan de manera muy similar; difieren principalmente en la forma en que se construyen.

##### Ejemplo: copia de archivos

Programa que emplea FileInputStream y FileOutputStream para copiar archivos CopiaArchivos, que utiliza flujos de bytes para **copiar un byte a la vez**.

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
public class CopiaArchivos {  
 public static void main(String[] args) throws IOException {  
  
 FileInputStream in = null;  
 FileOutputStream out = null;  
  
 try {  
 in = new FileInputStream("otto.txt");  
 out = new FileOutputStream("nohaycole.txt");  
 int c;  
  
 while ((c = in.read()) != -1) {  
 out.write(c);  
 }  
 } finally { // Hay que cerrar el flujo en cualquier condición.  
 if (in != null) {  
 in.close();  
 }  
 if (out != null) {  
 out.close();  
 }  
 }  
 }  
}

CopiaArchivos lee el flujo de entrada y escribe el flujo de salida, un byte a la vez.



* **El método read() devuelve un valor de byte en forma de un entero**, para poder emplear -1 como fin de flujo. **Cuando se alcanza el final del archivo, read() devuelve -1**.
* **El método write() escribe un byte en el flujo de salida**.
* **El método close() cierra el flujo**. Si no se cierra, el sistema operativo puede no liberar los recursos asociados con el archivo.
* Para **ficheros de texto (con caracteres, como en el ejemplo) es mejor emplear flujos de caracteres (character streams)**.
* Los flujos de bytes deben usarse **sólo para E/S más primitiva (binaria)**
* Todos los **otros tipos de flujos (incluso caracteres) se construyen sobre los flujos de bytes**.

Copia de archivos

CopiaArchivos parece un programa normal, pero en realidad **representa un tipo de E/S de bajo nivel que debería evitar**. Dado que otto.txt contiene datos de caracteres, **el mejor enfoque es usar flujos de caracteres**, como veremos más adelante. También hay flujos para tipos de datos más complejos. Los flujos de bytes **solo deben usarse para la E/S más primitiva**.

Entonces, ¿por qué hablar de flujos de bytes? **Porque todos los demás tipos de flujos se construyen sobre flujos de bytes**.

Ejercicio 1. Copia de archivos

Modifica el programa CopiaArchivos para que copie el archivo otto.txt en un archivo nohaycole.txt en la carpeta src/main/resources de tu proyecto.

Además, haz que el cierre de archivos se realice por medio de try-with-resources.

#### Cierre de flujos

**Cerrar un flujo cuando ya no se necesita es muy importante**. CopiaArchivos **utiliza un bloque finally para garantizar que ambos flujos se cierren incluso si se produce un error**. Esta práctica ayuda a evitar graves pérdidas de recursos.

La técnica más recomendada es **utilizar try-with-resources**, que permite que los flujos se cierren automáticamente al final del bloque try:

try (FileInputStream in = new FileInputStream("otto.txt");  
 FileOutputStream out = new FileOutputStream("nohaycole.txt")) {  
 int c;  
 while ((c = in.read()) != -1) {  
 out.write(c);  
 }  
}

Un **posible error es que CopiaArchivos no pudo abrir uno o ambos archivos**. Cuando esto sucede, la variable de flujo correspondiente al archivo nunca cambia desde su valor inicial nulo. Es por eso que CopiaArchivos se asegura de que cada variable de flujo contenga una referencia de objeto antes de llamar a close().

Cuando no usar flujos de bytes

CopiaArchivos parece un programa normal, pero en realidad **representa un tipo de E/S de bajo nivel que debería evitar**. Dado que otto.txt contiene datos de caracteres, **el mejor enfoque es usar flujos de caracteres**, como veremos más adelante. También hay flujos para tipos de datos más complejos. Los flujos de bytes **solo deben usarse para la E/S más primitiva**.

Entonces, ¿por qué hablar de flujos de bytes? **Porque todos los demás tipos de flujos se construyen sobre flujos de bytes**.

#### 1. InputStream

Flujos de entrada que [**heredan de InputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/InputStream.html), que es abstracta:

* [**ByteArrayInputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/ByteArrayInputStream.html): contiene un búfer interno que **contiene bytes que pueden ser leídos desde el flujo**. Un contador interno lleva un seguimiento del próximo byte que será suministrado por el método read. Cerrar un ByteArrayInputStream no tiene efecto. Los métodos en esta clase pueden ser llamados después de que el flujo haya sido cerrado sin generar una IOException.
* [FileInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/FileInputStream.html)
* [AudioInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/sound/sampled/AudioInputStream.html): es un flujo de entrada con un **formato de audio y longitud especificados**. La longitud se expresa en frames, no en bytes. Se proporcionan varios métodos para leer un cierto número de bytes del flujo, o un número no especificado de bytes. El flujo de entrada de audio lleva un seguimiento del último byte que se leyó. Puedes saltar sobre un número arbitrario de bytes para llegar a una posición posterior para la lectura. Un flujo de entrada de audio puede admitir marcas. Cuando estableces una marca, se recuerda la posición actual para que puedas volver a ella más tarde. **La** [**clase AudioSystem**](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/sound/sampled/AudioSystem.html) **incluye muchos métodos que manipulan objetos AudioInputStream**. Por ejemplo, los métodos te permiten:
  + Obtener un flujo de entrada de audio desde un archivo de audio externo, un flujo o una URL.
    - Escribir un archivo externo desde un flujo de entrada de audio.
    - Convertir un flujo de entrada de audio a un formato de audio diferente.
* [FilterInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/FileInputStream.html): encapsula otro flujo de entrada y proporciona funcionalidad adicional. Ejemplo:
  + [BufferedInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/BufferedInputStream.html): lee bytes de un flujo de entrada y los almacena en un búfer interno.
  + [DataInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/DataInputStream.html): lee primitivos de datos Java del flujo de entrada.
  + [PushbackInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/PushbackInputStream.html): permite que los bytes leídos se devuelvan al flujo de entrada.
* [**ObjectInputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/ObjectInputStream.html): **lee objetos Java serializados del flujo de entrada**.
* [PipedInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/PipedInputStream.html): implementa un tubo de entrada.
* [SequenceInputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/SequenceInputStream.html): concatena dos flujos de entrada.
* [~~StringBufferInputStream~~](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/StringBufferInputStream.html): desaprobada. Se recomienda el [uso de StringReader](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/StringReader.html).

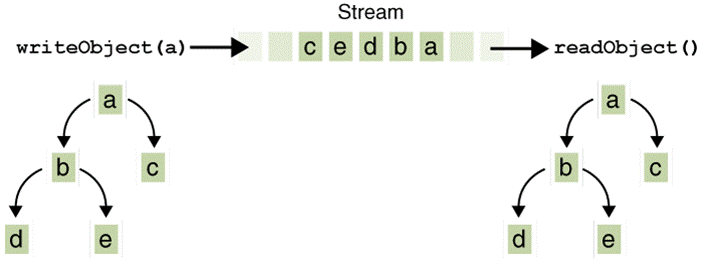
#### 2. OutputStream

Flujos de salida que [**heredan de OutputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/OutputStream.html), que es abstracta:

* [**ByteArrayOutputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/ByteArrayOutputStream.html): implementa un flujo de salida en el que los datos se escriben en un array de bytes. El búfer **crece automáticamente a medida que se escriben datos en él**. Los datos se pueden recuperar usando toByteArray() y toString(). **Cerrar a ByteArrayOutputStream no tiene ningún efecto**. Los métodos de esta clase se pueden llamar después de que se haya cerrado la secuencia sin generar un archivo IOException.
* [FileOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/FileOutputStream.html): flujo de salida para **escribir datos en un archivo File** o en un archivo FileDescriptor. El hecho de que un archivo esté disponible o pueda crearse depende de la plataforma subyacente. Algunas plataformas, en particular, permiten que un archivo sea abierto para escritura por solo uno FileOutputStream (u otro objeto de escritura de archivos) a la vez. En tales situaciones, los constructores de esta clase fallarán si el archivo involucrado ya está abierto. FileOutputStream está destinado a escribir **flujos de bytes sin formato, como datos de imágenes**. Para escribir secuencias de caracteres debe usarse el orientado a carácter FileWriter.
* [**ObjectOutputStream**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/ObjectOutputStream.html)**: escribe objetos Java serializados en un flujo de salida.**
* [PipedOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/PipedOutputStream.html): implementa un tubo de salida.
* [FilterOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/FilterOutputStream.html): encapsula otro flujo de salida y proporciona funcionalidad adicional. Ejemplo:
  + [BufferedOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/BufferedOutputStream.html): escribe bytes en un flujo de salida y los almacena en un búfer interno.
  + [PrintStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/PrintStream.html): proporciona métodos para imprimir representaciones de datos primitivos y objetos en un flujo de salida. un ejemplo de uso es System.out.
  + [CheckedOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/zip/CheckedOutputStream.html): calcula un valor de comprobación de suma de verificación (checksum)para los datos escritos en el flujo de salida. Se puede emplear para comprobar la integridad de los datos de salida.
  + [CipherOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/javax/crypto/CipherOutputStream.html): escribe datos cifrados en un flujo de salida. Está compuesto por **un OutputStream y un objeto de tipo Cipher**, para procesar los datos antes de escribirlos en el flujo de salida. Debe ser inicializado con un modo de cifrado y una clave.
  + [DataOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/DataOutputStream.html): escribe **datos primitivos Java en el flujo de salida**. Los datos se pueden recuperar usando DataInputStream.
  + [DeflaterOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/zip/DeflaterOutputStream.html): comprime los datos escritos en el flujo de salida. Tiene dos subclases: GZIPOutputStream y ZipOutputStream.
    - [ZipOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/zip/ZipOutputStream.html): escribe archivos ZIP.
    - [GZIPOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/zip/GZIPOutputStream.html): escribe archivos GZIP.
  + [DigestOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/security/DigestOutputStream.html): calcula un resumen de mensaje de los datos escritos en el flujo de salida. Se puede emplear para comprobar la integridad de los datos de salida.
  + [InflaterOutputStream](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/zip/InflaterOutputStream.html): implanta un filtro de flujo de salida para descomprimir datos comprimidos en formato de compresión de “deflate”.

#### 3. ObjectInputStream y ObjectOutputStream

[**ObjectInputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/ObjectInputStream.html): lee objetos Java serializados del flujo de entrada y los deserializa. [**ObjectOutputStream**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/ObjectOutputStream.html): escribe objetos Java serializados en un flujo de salida.



Para emplear las clases ObjectInputStream, ObjectOutStream los objetos a leer (escribir **deben implantar la interface: Serializable** (dicha interface no tiene métodos para implantar)

Para escribir:

Object ob = new Object();  
out.writeObject(ob); //out es un flujo de tipo ObjectOutputStream  
out.writeObject(ob);

Para leer:

Object ob1 = in.readObject();  
Object ob2 = in.readObject();

Serialización

La **serialización** es el proceso de convertir un objeto en una secuencia de bytes que se pueden escribir en un flujo de salida y, posteriormente, **reconstruir el objeto a partir de esos bytes**. La **deserialización** es el proceso inverso: **reconstruir un objeto a partir de una secuencia de bytes**.

Ejercicio 2. Serialización

Crea una clase Persona con los atributos nombre y edad. Crea un programa que serialice y deserialice un objeto de tipo Persona.

Debe tener un menú con las siguientes opciones:

1. Añadir persona.
2. Mostrar personas.
3. Buscar persona (por número o por nombre, según consideres)
4. Salir

Puedes hacerlo desde consola o por medio de una interfaz gráfica, haciendo uso de JOptionPane para introducir los datos ([JOptionPane.showInputDialog](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/swing/JOptionPane.html#showInputDialog(java.lang.Object))) y mostrar los resultados ([JOptionPane.showMessageDialog](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/swing/JOptionPane.html#showMessageDialog(java.awt.Component,java.lang.Object))).

Ejercicio 3. Serialización de colecciones

Crea una clase ColeccionPersonas que contenga una colección de objetos de tipo Persona. Implementa la interface Serializable y crea un programa que serialice y deserialice un objeto de tipo ColeccionPersonas.

#### 4. Lectura desde URL

Para leer desde una URL, se puede emplear la clase URL y openStream():

import java.io.\*;  
  
public class LeerURL {  
 public static void main(String[] args) throws Exception {  
 // URL url = new URL("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/"); // Desaprobado.  
 // Versión actualizada:  
 URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/");  
 URL url = uri.toURL();  
   
 try (InputStream is = url.openStream();  
 InputStreamReader isr = new InputStreamReader(is); // es un puente de bytes a caracteres.  
 int c;  
 while ((c = isr.read()) != -1) {  
 System.out.print((char) c);  
 }  
 }  
// // Código equivalente con buffer:  
// try (InputStream is = url.openStream();  
// InputStreamReader isr = new InputStreamReader(is);  
// BufferedReader br = new BufferedReader(isr)) { // Lo veremos en el siguiente apartado.  
// String line;  
// while ((line = br.readLine()) != null) {  
// System.out.println(line);  
// }  
// }  
 }  
}

##### URI/URL

La clase URL tiene constructores desaprobados, **se recomienda emplear URI para crear una URL**:

URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/");  
 URL url = uri.toURL();

url.openStream(); // Abreviatura de:  
 url.openConnection().getInputStream(); // openConnection() devuelve un objeto de tipo URLConnection.

// Implantación de openStream() en la clase URL:  
public final InputStream openStream() throws java.io.IOException {  
 return openConnection().getInputStream();  
}

Los constructores de URL está desaprobada, se recomienda emplear URI:

URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/");  
 URL url = uri.toURL();

##### URLConnection

El método openConnection() de URL devuelve un objeto de tipo URLConnection:

URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/");  
 URL url = uri.toURL();  
 URLConnection urlConnection = url.openConnection();  
 urlConnection.getInputStream();

##### HttpURLConnection

Permite añadir elementos específicos de HTTP, como el tamaño del contenido, o el tipo de archivo:

URL url = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam/ad/").toURL();  
 HttpURLConnection httpConnection = (HttpURLConnection) url.openConnection(); // Hereda de URLConnection  
 httpConnection.getInputStream();  
 httpConnection.setRequestMethod("HEAD");  
 long tamanho = httpConnection.getContentLengthLong();

Ejercicio 4. Lectura de URL

Crea un programa que lea el contenido de una URL y lo muestre por pantalla.

Mejore el programa para que p**ida una URL y la guarde en un archivo en una carpeta seleccionada del disco mediante un JFileChooser.**

¿Serías capaz de **mostrar el tamaño del contenido de la URL**? ¿Y que ponga la **extensión adecuada al archivo**?

Ayuda: Puedes emplear HttpURLConnection para obtener el tamaño del contenido y para obtener el Content-Type puedes emplear el método getContentType().

Ejercicio 5. Lectura de URL con HttpURLConnection

Amplía el ejercicio anterior para que emplee HttpURLConnection y **muestre la información de la cabecera HTTP**.

## 01.05 FLUJOS DE CARACTERES

* [Flujos de caracteres (Character Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0105characterstream/#flujos-de-caracteres-character-streams)
  + [1. Reader y Writer](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0105characterstream/#1-reader-y-writer)
  + [2. Lectura de líneas completas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0105characterstream/#2-lectura-de-l%C3%ADneas-completas)
  + [3. Diagrama de clases de Reader Java:](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0105characterstream/#3-diagrama-de-clases-de-reader-java)

### Flujos de caracteres (Character Streams)

* Los flujos de caracteres **leen/escriben datos de texto** y tienen nombres de clase que terminan en Reader o Writer.
* Automáticamente, **transforma caracteres Unicode (formato de Java) al conjunto de caracteres local**.
* Todas **las clases descienden de Reader y Writer**.
* Hay **muchas clases de flujos de carácter, como:** [**FileReader**](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/FileReader.html) **(usa internamente FileInputStream)**, FileWriter (usa internamente **FileOutpuStream**). Todos los restantes flujos funcionan de igual modo, sólo difieren en la forma de construirlos.

**Java almacena valores de caracteres utilizando convenciones Unicode.** La E/S de **flujos de caracteres traduce automáticamente este formato interno hacia y desde el conjunto de caracteres local**. En locales occidentales, como el juego de caracteres Latin-1 o Windows-1252, el conjunto de caracteres local es generalmente un superconjunto de ASCII de 8 bits. En locales asiáticos, el conjunto de caracteres local es un conjunto de caracteres de doble byte.

En la E/S con flujos de caracteres, **la entrada y salida realizada con clases de flujo se traduce automáticamente hacia y desde el conjunto de caracteres local**. Un programa que utiliza flujos de caracteres en lugar de flujos de bytes **se adapta automáticamente al conjunto de caracteres local y está listo para la internacionalización, todo sin esfuerzo adicional por parte del programador**.

Si la internacionalización no es prioritario, **puedes usar las clases de flujos de caracteres sin prestar mucha atención a los problemas de conjunto de caracteres**. Más tarde, si la internacionalización se convierte en una prioridad, tu programa puede adaptarse sin una recodificación extensa (existen constructores y métodos que recogen el juego de caracteres).

#### 1. Reader y Writer

**Todas las clases de flujos de caracteres heredan de la** [**clase abstracta Reader**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/Reader.html) **y la** [**clase abstracta Writer**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/Writer.html). Al igual que con los flujos de bytes, existen clases de flujos de caracteres que se especializan en **la E/S de archivos:** [**FileReader**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FileReader.html) **y** [**FileWriter**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/FilterWriter.html). El ejemplo CopiarCaracteres ilustra estas clases.

import java.io.FileReader;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class CopiarCaracteres {  
 public static void main(String[] args) throws IOException {  
  
 FileReader inputStream = null;  
 FileWriter outputStream = null;  
  
 try {  
 inputStream = new FileReader("otto.txt");  
 outputStream = new FileWriter("nohaycole.txt");  
  
 int c;  
 while ((c = inputStream.read()) != -1) {  
 outputStream.write(c);  
 }  
 } finally {  
 if (inputStream != null) {  
 inputStream.close();  
 }  
 if (outputStream != null) {  
 outputStream.close();  
 }  
 }  
 }  
}

CopiarCaracteres es muy similar a CopiaArchivos. La diferencia más importante es que **CopiarCaracteres ==utiliza FileReader y FileWriter para entrada y salida en lugar de FileInputStream y FileOutputStream**==. Observa que **tanto CopiaArchivos como CopiarCaracteres emplean una variable int para leer y escribir**. Sin embargo, **en CopiarCaracteres, la variable int contiene un valor de carácter en sus últimos 16 bits**; **en CopiaArchivos, la variable int contiene un valor de byte en sus últimos 8 bits**.

Con try-with-resources, el código es más limpio y más fácil de leer. **FileReader y FileWriter se cierran automáticamente** cuando el bloque try-with-resources se completa:

try (  
 FileReader inputStream = new FileReader("otto.txt");  
 FileWriter outputStream = new FileWriter("nohaycole.txt");  
 ) {  
 int c;  
 while ((c = inputStream.read()) != -1) {  
 outputStream.write(c);  
 }  
 }

##### Flujos de caracteres que utilizan flujos de bytes

Los **flujos de caracteres suelen ser “envoltorios” para flujos de bytes**. El **flujo de caracteres utiliza el flujo de bytes para realizar la E/S física, mientras que el flujo de caracteres maneja la traducción entre caracteres y bytes**. **FileReader, por ejemplo, utiliza FileInputStream, mientras que FileWriter utiliza FileOutputStream**.

**InputStreamReader** y **OutputStreamWriter**

Son flujos de caracteres que leen y escriben bytes, respectivamente, son **flujos de “puente” byte-a-carácter de propósito general:** [**InputStreamReader**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/InputStreamReader.html) **y** [**OutputStreamWriter**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/OutputStreamWriter.html).

Se emplean para crear flujos de caracteres **cuando no haya clases de flujo de caracteres preempaquetadas** que cumplan con las necesidades. Por ejemplo, para **crear flujos de caracteres a partir de los flujos de bytes** proporcionados por las clases de **Socket**, como se muestra en el siguiente ejemplo:

import java.io.\*;  
import java.net.\*;  
  
public class ClienteEcho {  
 public static void main(String[] args) throws IOException {  
  
 if (args.length != 2) {  
 System.err.println("Uso: java ClienteEcho <nombre host> <número puerto>");  
 System.exit(1);  
 }  
  
 String nombreHost = args[0];  
 int numeroPuerto = Integer.parseInt(args[1]);  
  
 try (Socket echoSocket = new Socket(nombreHost, numeroPuerto); // Socket es un flujo de bytes  
 PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true); // PrintWriter es un flujo de caracteres que envía datos a un flujo de bytes. true para autoflush. Escribirá en el flujo de bytes cada vez que se llame a println  
 BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream())); // InputStreamReader es un puente byte a carácter, leemos bytes del flujo de bytes y los convertimos a caracteres  
 BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in)) // InputStreamReader es un puente byte a carácter  
 ) {  
 String entradaUsuario;  
 while ((entradaUsuario = stdIn.readLine()) != null) {  
 out.println(entradaUsuario); // envía la entrada del usuario al servidor  
 System.out.println("echo: " + in.readLine());  
 }  
 } catch (UnknownHostException e) {  
 System.err.println("Host desconocido " + nombreHost);  
 System.exit(1);  
 } catch (IOException e) {  
 System.err.println("NO ha sido posible establecer la conexión con " +  
 nombreHost);  
 System.exit(1);  
 }  
 }  
}

O para **lectura desde teclado y salida a consola**:

import java.io.\*;  
  
public class EjemploPuente {  
 public static void main(String[] args) throws IOException {  
 try (  
 Reader reader = new InputStreamReader(System.in);  
 Writer writer = new OutputStreamWriter(System.out);  
 ) {  
 int c;  
 while ((c = reader.read()) != -1) {  
 writer.write(c);  
 }  
 }  
 }  
}

#### 2. Lectura de líneas completas

La E/S de caracteres suele ocurrir en unidades más grandes que los caracteres individuales. Una **unidad común es la línea**: una cadena de caracteres **con un terminador de línea al final**.

Terminadores de línea

Un **terminador de línea puede ser una secuencia de retorno de carro/avance de línea ("\r\n")**, un solo retorno de carro ("\r"), o un solo avance de línea ("\n"). Admitir todos los terminadores de línea posibles permite a los programas leer archivos de texto creados en cualquiera de los sistemas operativos ampliamente utilizados.

**En Windows, el terminador de línea es “\r\n”. En Unix, el terminador de línea es “\n”. En Macintosh, el terminador de línea es “\r**”.

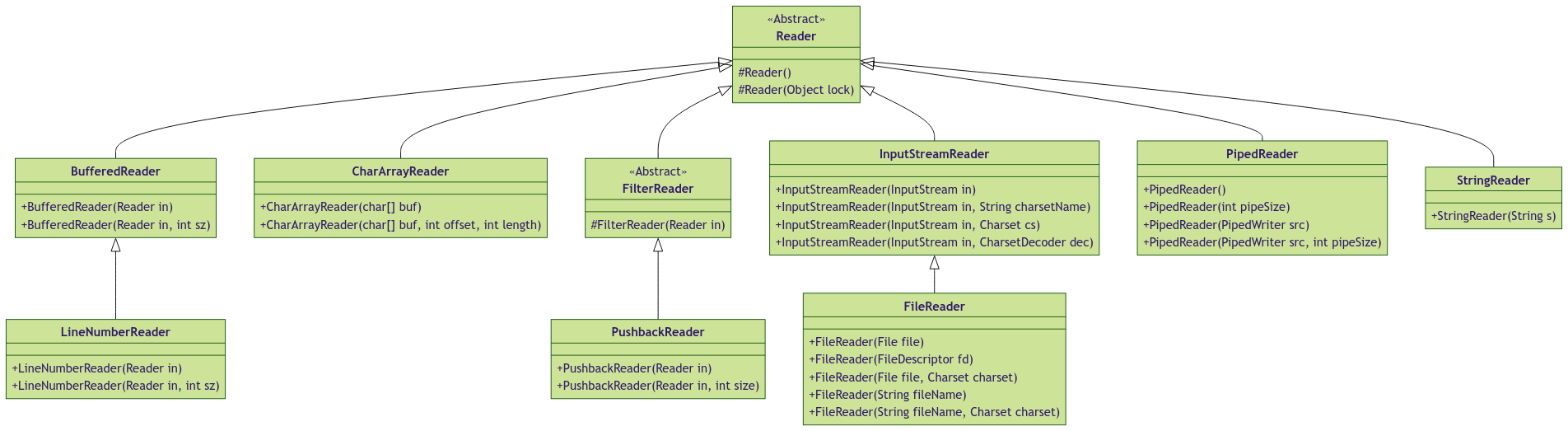
Modifiquemos el ejemplo CopiarCaracteres para usar E/S **orientada a líneas**. Para hacer esto, tenemos que usar dos clases **con buffer o memoria internmedia** (que guarda los caracteres de toda la línea, o más), BufferedReader y PrintWriter. Veresmos estas clases con mayor profundidad en E/S en el siguiente apartado. El ejemplo CopiarCaracteres **invoca BufferedReader.readLine y PrintWriter.println para realizar la entrada y salida una línea a la vez**.

import java.io.FileReader;  
import java.io.FileWriter;  
import java.io.BufferedReader;  
import java.io.PrintWriter;  
import java.io.IOException;  
  
public class CopiaLineas {  
 public static void main(String[] args) throws IOException {  
  
 BufferedReader inputStream = null;  
 PrintWriter outputStream = null;  
  
 try {  
 inputStream = new BufferedReader(new FileReader("otto.txt"));  
 outputStream = new PrintWriter(new FileWriter("nohaycole.txt"));  
  
 String l;  
 while ((l = inputStream.readLine()) != null) {  
 outputStream.println(l);  
 }  
 } finally {  
 if (inputStream != null) {  
 inputStream.close();  
 }  
 if (outputStream != null) {  
 outputStream.close();  
 }  
 }  
 }  
}

Invocar **readLine devuelve una línea de texto con la línea**. CopiaLineas genera cada línea usando println, que añade el terminador de línea para el sistema operativo actual. Esto puede que no sea el mismo terminador de línea que se usó en el archivo de entrada.

Hay muchas maneras de estructurar la entrada y salida de texto más allá de caracteres y líneas.

#### 3. Diagrama de clases de Reader Java:



La API a menudo incluye clases similares tanto para flujos de bytes como para flujos de caracteres, como FileInputStream y FileReader. La **diferencia entre las dos clases se basa en cómo se leen o escriben los bytes en el flujo**.

Es importante recordar que, **aunque los flujos de caracteres no contienen la palabra “Stream” en su nombre de clase, siguen siendo flujos de E/S**. El uso de **“Reader/Writer” en el nombre es simplemente para distinguirlas de los flujos de bytes**.

Los **flujos de bytes se utilizan principalmente para trabajar con datos binarios**, como una imagen o un archivo ejecutable, mientras que los **flujos de caracteres se utilizan para trabajar con archivos de texto**. Dado que las clases de flujos de bytes pueden escribir todo tipo de datos binarios, incluidas cadenas, se deduce que las clases de flujos de caracteres no son estrictamente necesarias. Sin embargo, **existen ventajas en usar las clases de flujos de caracteres, ya que se centran específicamente en la gestión de datos de caracteres y cadenas**. Por ejemplo, puedes **emplear una clase Writer para escribir un valor de cadena en un archivo sin necesidad de preocuparte por la codificación de caracteres subyacente del archivo**.

La ***codificación de caracteres*** determina cómo se codifican y almacenan los caracteres en bytes en un flujo y cómo se leen posteriormente o decodifican como caracteres. Aunque esto puede parecer sencillo, **Java admite una amplia variedad de codificaciones de caracteres, desde aquellas que pueden utilizar un byte para caracteres latinos, como UTF-8 y ASCII**, hasta aquellas que utilizan dos o más bytes por carácter, **como UTF-16**. No es necesario entrar en detalle sobre las codificaciones de caracteres, pero debes estar familiarizado con sus nombres si te encuentras con ellos algún día y saber por dónde van los tiros ;-).

Flujo de caracteres para texto

En cuanto a la codificación de caracteres, simplemente recuerda que **usar un flujo de caracteres es mejor para trabajar con datos de texto que un flujo de bytes**. Las clases de flujos de caracteres se crearon por conveniencia, y debes aprovecharlas cuando sea posible.

## 01.06 FLUJOS DE E/S CON BUFFER

* [1. Flujos de Bajo Nivel vs. flujos de alto nivel](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#1-flujos-de-bajo-nivel-vs-flujos-de-alto-nivel)
  + [1.1. Flujos de bajo nivel (sin buffer)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#11-flujos-de-bajo-nivel-sin-buffer)
  + [1.2. Flujos de alto nivel (con buffer)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#12-flujos-de-alto-nivel-con-buffer)
* [2. Clases base para flujos: InputStream, OutputStream, Reader y Writer](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#2-clases-base-para-flujos-inputstream-outputstream-reader-y-writer)
  + [2.1. Identificación de clases de E/S con flujos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#21-identificaci%C3%B3n-de-clases-de-es-con-flujos)
* [3. Tabla resumen de clases de flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#3-tabla-resumen-de-clases-de-flujos-de-es)

### 1. Flujos de Bajo Nivel vs. flujos de alto nivel

Otra forma de familiarizarse con la API java.io es **dividir los flujos en flujos de bajo nivel y flujos de alto nivel (con buffer o memoria intermedia)**.

#### 1.1. Flujos de bajo nivel (sin buffer)

Un **flujo de bajo nivel (sin buffer) se conecta directamente a la fuente de datos**, como un archivo, un array o un String. Los flujos de bajo nivel **procesan los datos o recursos en bruto** y se acceden de manera directa y sin filtrar.

Por ejemplo, una FileInputStream es una clase que lee datos de archivos de un byte a la vez.

En los flujos **sin buffer cada petición de lectura/escritura se envía directamente al sistema E/S: puede ser ineficiente (acceso a disco, actividad de red,…)**

#### 1.2. Flujos de alto nivel (con buffer)

Por otro lado, un **flujo de alto nivel se construye sobre un flujo mediante el encapsulamiento**. La *encapsulación* es el proceso mediante el cual una instancia se pasa al constructor de otra clase, y las operaciones en la instancia resultante se filtran y aplican a la instancia original.

Por ejemplo, echa un vistazo a los objetos FileReader y BufferedReader en el siguiente código de ejemplo:

try (var br = new BufferedReader(new FileReader("noHayCole.txt"))) {  
 System.out.println(br.readLine());  
}

En este ejemplo, **FileReader es el flujo de bajo nivel para la lectura**, mientras que **BufferedReader es el flujo de alto nivel que toma un FileReader como entrada**. Muchas operaciones en el flujo de alto nivel pasan como operaciones a el flujo de bajo nivel subyacente, como read() o close(). Otras operaciones anulan o agregan nueva funcionalidad a los métodos de el flujo de bajo nivel.

**Un flujo de buffer puede agregar nuevos métodos, como readLine(), así como mejoras de rendimiento para leer y filtrar los datos de bajo nivel**.

Los flujos de alto nivel **pueden tomar otros flujos de alto nivel como entrada**. Por ejemplo, aunque el siguiente código pueda parecer un poco extraño al principio, el estilo de encapsular un flujo es bastante común en la práctica:

try (var ois = new ObjectInputStream(new BufferedInputStream(  
 new FileInputStream("noHayCole.txt")))) {  
 System.out.print(ois.readObject());  
}

En este ejemplo, FileInputStream es el flujo de bajo nivel que interactúa directamente con el archivo, la cual está envuelta por BufferedInputStream de alto nivel para mejorar el rendimiento. Finalmente, el objeto completo está envuelto por ObjectInputStream, de alto nivel, que nos permite interpretar los datos como un objeto Java.

Las únicas clases de flujos de bajo nivel con las que debes estar familiarizado son las que operan en archivos. **El resto de las clases de flujos no abstractas son todas flujos de alto nivel**.

Utiliza flujos con búfer al trabajar con archivos

Como se comentó brevemente, las **clases con “Buffered” leen o escriben datos en bloques en lugar de un solo byte o carácter a la vez**. La **mejora de rendimiento** al utilizar una clase con búfer para acceder a un flujo de bajo nivel de archivos no se puede exagerar. A menos que estés haciendo algo muy especializado en tu aplicación, **siempre debes envolver un flujo de archivo con una clase con búfer en la práctica**.

Una de las razones por las que los flujos con búfer tienden a funcionar tan bien en la práctica es que **muchos sistemas de archivos están optimizados para el acceso secuencial al disco**. Cuantos más bytes secuenciales leas a la vez, menos viajes de ida y vuelta entre el proceso Java y el sistema de archivos, lo que mejora el acceso de tu aplicación. Por ejemplo, acceder a 1,600 bytes secuenciales es mucho más rápido que acceder a 1,600 bytes dispersos por el disco duro.

### 2. Clases base para flujos: InputStream, OutputStream, Reader y Writer

La **biblioteca java.io define cuatro clases abstractas que son las clases base de todas las clases de flujos definidas en la API**:

* InputStream
* OutputStream
* Reader
* Writer

Frecuentemente, los constructores de flujos de alto nivel toman una referencia de la clase abstracta. Por ejemplo, BufferedWriter toma un objeto Writer como entrada, lo que le permite tomar cualquier subclase de Writer.

Es un error común para iniciados mezclar y combinar clases de flujos que no son compatibles entre sí. Por ejemplo, echa un vistazo a cada uno de los siguientes ejemplos y ve si puedes determinar por qué **no se compilan**:

new BufferedInputStream(new FileReader("z.txt")); // NO COMPILA por mezclar clases de Reader con clases de InputStream  
  
new BufferedWriter(new FileOutputStream("z.txt")); // NO COMPILA por mezclar clases de Writer con clases de OutputStream  
  
new ObjectInputStream(new FileOutputStream("z.txt")); // NO COMPILA por mezclar clases de InputStream con clases de OutputStream  
  
new BufferedInputStream(new InputStream()); // NO COMPILA porque InputStream es una clase abstracta

Los primeros dos ejemplos no se compilan porque mezclan clases de Reader/Writer con clases de InputStream/OutputStream, respectivamente. El tercer ejemplo no se compila porque estamos mezclando una OutputStream con una InputStream. Aunque es posible leer datos de una InputStream y escribirlos en una OutputStream, envolver el flujo no es la forma de hacerlo.

Como veremos más adelante, los datos **deben copiarse, a menudo de manera iterativa**. Finalmente, el último ejemplo no se compila porque InputStream es una clase abstracta y, por lo tanto, no puedes crear una instancia de ella.

#### 2.1. Identificación de clases de E/S con flujos

Presta atención al **nombre de la clase de E/S**, ya que descifrarlo a menudo te proporciona pistas de contexto sobre **lo que hace la clase**. Por ejemplo, sin necesidad de buscarlo, debería estar claro que FileReader es una clase que lee datos de un archivo como caracteres o cadenas. Además, ObjectOutputStream parece una clase que escribe datos de objeto en un flujo de bytes.

**Revisión de las Propiedades de los Nombres de Clase de java.io**

* Una clase con las palabras "**InputStream**" u “**OutputStream**” en su nombre se utiliza para **leer o escribir datos binarios** (o de bytes), respectivamente.
* Una clase con las palabras "**Reader**" o “**Writer**” en su nombre se utiliza para **leer o escribir datos de caracteres (o cadenas)**, respectivamente.
* La **mayoría**, pero no todas, **las clases de entrada tienen una clase de salida** correspondiente (FileInputStream y FileOutputStream, por ejemplo)
* Un **flujo de bajo nivel se conecta directamente a la fuente de datos** (FileInputStream y FileOutputStream, por ejemplo):

FileReader in = new FileReader("unaVacaLoca.mp3");

* Un **flujo de buffer se construye sobre otro flujo de bajo** mediante encapsulación (dentro de un buffer):

BufferedReader in = new BufferedReader(new FileReader("chocolateCaramelo.mp3"));

* Una clase con "**Buffered**" en su nombre **lee o escribe datos** en grupos de bytes o caracteres de una **memoria intermedia o buffer** y, a menudo, **mejora el rendimiento en sistemas de archivos secuenciales**.

*Con algunas excepciones, sólo envuelves un flujo con otro flujo si comparten el mismo padre abstracto (FileReader puede ser encapsulado en un BufferedReader, por ejemplo), salvo clases que pasan flujos de bytes (InputStream) en caracteres (Reader), por ejemplo:* [*InputStreamReader*](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/InputStreamReader.html)

### 3. Tabla resumen de clases de flujos de E/S

[Tabla 1](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#_bookmark7) y [Tabla 2](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#_bookmark9) se muestran las clases base abstractas de flujos y las clases concretas de flujos de E/S que debes conocer. Ten en cuenta que **la mayoría de la información sobre cada flujo, como si es de entrada o salida o si accede a datos mediante bytes o caracteres, se puede deducir solo por el nombre**.

[Tabla 1](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#_bookmark7) **Las clases base abstractas de flujos de E/S de java.io**:

|  |  |
| --- | --- |
| **Clase** | **Descripción** |
| **InputStream** | Clase abstracta para todas los flujos de entrada de bytes |
|  |  |
| **OutputStream** | Clase abstracta para todas los flujos de salida de bytes |
|  |  |
| **Reader** | Clase abstracta para todas los flujos de entrada de caracteres |
|  |  |
| **Writer** | Clase abstracta para todas los flujos de salida de caracteres |

[Tabla 2](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0106javaiobuffer/#_bookmark9) **Clases implemementadas de flujos de E/S de java.io** que debes conocer:

|  |  |  |
| --- | --- | --- |
| **Clase** | **Bajo/Alto Nivel** | **Descripción** |
| FileInputStream | Bajo | Lee datos de archivos como bytes |
| FileOutputStream | Bajo | Escribe datos de archivos como bytes |
| FileReader | Bajo | Lee datos de archivos como caracteres |
| FileWriter | Bajo | Escribe datos de archivos como caracteres |
| BufferedInputStream | Alto | Lee datos de bytes de un flujo de entrada existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento |
| BufferedOutputStream | Alto | Escribe datos de bytes en un flujo de salida existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento |
| BufferedReader | Alto | Lee datos de caracteres de un objeto Reader existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento |
| BufferedWriter | Alto | Escribe datos de caracteres en un objeto Writer existente de manera bufferizada, lo que mejora la eficiencia y elrendimiento |
| ObjectInputStream | Alto | Deserializa tipos de datos primitivos de Java y gráficos de objetos de Java a partir de un flujo de entrada existente |
| ObjectOutputStream | Alto | Serializa tipos de datos primitivos de Java y gráficos de objetos de Java en un flujo de salida existente |
| **PrintStream** | Alto | **Escribe** **representaciones formateadas de objetos Java en un flujo binario** |
| **PrintWriter** | Alto | **Escribe** representaciones **formateadas de objetos Java en un flujo de caracteres** |

## 01.07 OPERACIONES COMUNES CON FLUJOS DE E/S.

* [1. Operaciones con Flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#1-operaciones-con-flujos-de-es)
  + [1.1. Lectura y escritura de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#11-lectura-y-escritura-de-datos)
  + [1.2. Cierre de flujos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#12-cierre-de-flujos)
  + [1.3. Cierre de flujos envueltos en otro flujo (con buffer)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#13-cierre-de-flujos-envueltos-en-otro-flujo-con-buffer)
  + [1.4. Manipulación de flujos de entrada: Mark, Reset y Skip](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#14-manipulaci%C3%B3n-de-flujos-de-entrada-mark-reset-y-skip)
    - [*mark() y reset()*](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#mark-y-reset)
    - [*skip()*](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#skip)
  + [1.5. Flushing de flujos de salida (Output Streams)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#15-flushing--de-flujos-de-salida-output-streams)
* [2. Resumen de métodos más comunes de flujos de E/S](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0107javaistream/#2-resumen-de-m%C3%A9todos-m%C3%A1s-comunes-de-flujos-de-es)

### 1. Operaciones con Flujos de E/S

Aunque existen muchas clases de flujos, muchas de ellas comparten las mismas operaciones. En esta sección, revisaremos los **métodos comunes entre varias clases de flujos**. En la siguiente sección, cubriremos clases de flujos específicas.

#### 1.1. Lectura y escritura de Datos

Los flujos de E/S se tratan de leer y escribir datos, por lo que no debería sorprendernos que los métodos más importantes sean read() y write(). Tanto InputStream como Reader declaran el siguiente método para leer datos de bytes de un flujo:

// InputStream y Reader  
  
public int read() throws IOException

Del mismo modo, OutputStream y Writer definen el siguiente método para escribir un byte en el flujo:

// OutputStream y Writer  
  
public void write(int b) throws IOException

Espera un momento. Dijimos que estamos leyendo y escribiendo bytes, ¿entonces por qué los métodos usan int en lugar de byte? Recuerda, **el tipo de dato byte tiene un rango de 256 caracteres. Se necesitaba un valor adicional para indicar el final de un flujo**. Los autores de Java decidieron usar un tipo de dato más grande, **int, para que valores especiales como -1 indiquen el final de un flujo**. Las clases de flujos de salida también utilizan int para ser coherentes con las clases de flujos de entrada.

// Ejemplo de métodos copyStream() que leen desde un InputStream o Reader  
// y escriben en un OutputStream o Writer, respectivamente. En ambos ejemplos,  
// -1 se usa para indicar el final del flujo.  
  
void copyStream(InputStream in, OutputStream out) throws IOException {  
 int b;  
  
 while ((b = in.read()) != -1) {  
 out.write(b);  
 }  
}  
  
void copyStream(Reader in, Writer out) throws IOException {  
 int b;  
  
 while ((b = in.read()) != -1) {  
 out.write(b);  
 }  
}

Las **clases de flujos de bytes también incluyen métodos sobrecargados para leer y escribir múltiples bytes a la vez**.

// InputStream  
  
public int read(byte[] b) throws IOException  
  
public int read(byte[] b, int offset, int length) throws IOException  
  
// OutputStream  
  
public void write(byte[] b) throws IOException  
  
public void write(byte[] b, int offset, int length) throws IOException

Los **valores de offset y length se aplican al array en sí**. Por ejemplo, un offset de 5 y una longitud de 3 indican que el flujo debería leer hasta 3 bytes de datos y colocarlos en el array comenzando desde la posición 5.

Existen **métodos equivalentes para las clases de flujos de caracteres que usan char en lugar de byte**.

// Reader  
  
public int read(char[] c) throws IOException  
  
public int read(char[] c, int offset, int length) throws IOException  
  
// Writer  
  
public void write(char[] c) throws IOException  
  
public void write(char[] c, int offset, int length) throws IOException

#### 1.2. Cierre de flujos

Todos los flujos de E/S incluyen un método para **liberar cualquier recurso dentro del flujo cuando ya no se necesita**.

// Todas las clases de flujos de E/S  
  
public void close() throws IOException

Dado que **los flujos se consideran recursos, es fundamental que todos los flujos de E/S se cierren después de su uso, para evitar posibles fugas de recursos**.

Dado que **todos los flujos de E/S implementan la interfaz Closeable**, **la mejor manera de hacerlo es con una declaración try-with-resources**.

try (var fis = new FileInputStream("datos.txt")) {  
 System.out.print(fis.read());  
}

En muchos sistemas de archivos, **no cerrar un archivo correctamente podría dejarlo bloqueado por el sistema operativo, impidiendo que otros procesos lo lean o escriban hasta que el programa se termine**. EN la medida de lo posible, **cerraremos los recursos del flujo usando la sintaxis de try-with-resources**, ya que esta es la forma preferida de cerrar recursos en Java. También **utilizaremos var para acortar las declaraciones** , ya que estas declaraciones pueden volverse bastante largas (en el aula suelo poner el nombre de clase para poner el tipo concreto y que lo conozcáis, pero es **mejor hacerlo con var**).

¿Y si necesitas pasar un flujo a un método? Eso está bien, pero el **flujo debe cerrarse en el método que lo creó**.

public void printData(InputStream is) throws IOException {  
 int b;  
  
 while ((b = is.read()) != -1) {  
 System.out.print(b);  
 }  
}  
  
public void readFile(String fileName) throws IOException {  
 try (var fis = new FileInputStream(fileName)) {  
 printData(fis);  
 }  
}

En este ejemplo, el flujo se crea y se cierra en el método readFile(), mientras que printData() procesa su contenido.

#### 1.3. Cierre de flujos envueltos en otro flujo (con buffer)

Cuando trabajas con un flujo envuelto (con buffer), **solo necesitas usar close() en el objeto superior.** Al hacerlo, **se cerrarán los flujos subyacentes**.

El siguiente ejemplo es válido y resultará en tres llamadas separadas a close(), pero es innecesario:

try (var fis = new FileOutputStream("zoo-banner.txt");  
 // Innecesario  
 var bis = new BufferedOutputStream(fis);  
 var ois = new ObjectOutputStream(bis)) {  
 ois.writeObject("Hola");  
 }

En cambio, podemos confiar en que ObjectOutputStream cierre BufferedOutputStream y FileOutputStream. Lo siguiente llamará solo a un método close() en lugar de tres:

try (var ois = new ObjectOutputStream(  
 new BufferedOutputStream(  
 new FileOutputStream("zoo-banner.txt")))) {  
 ois.writeObject("Hola");  
 }

#### 1.4. Manipulación de flujos de entrada: Mark, Reset y Skip

Todas las clases de flujos de entrada incluyen los siguientes métodos para **manipular el orden en el que se leen los datos de un flujo**:

// InputStream y Reader  
  
public boolean markSupported();  
  
public void mark(int readLimit);  
  
public void reset() throws IOException;  
  
public long skip(long n) throws IOException;

Los métodos **mark() y reset() devuelven un flujo a una posición anterior**.

Antes de llamar a cualquiera de estos métodos, debes llamar al método **markSupported(), que devuelve true solo si mark() es compatible**.

El **método skip()** es bastante simple; básicamente, **lee datos del flujo y descarta el contenido**.

##### mark() y reset()

Supongamos que tenemos una instancia de InputStream cuyos próximos valores son “LEON”. Considera el siguiente fragmento de código:

public void readData(InputStream is) throws IOException {  
 System.out.print((char) is.read()); // L  
 if (is.markSupported()) {  
 is.mark(100); // Marca hasta 100 bytes  
 System.out.print((char) is.read()); // E  
 System.out.print((char) is.read()); // O  
 is.reset(); // Restablece el flujo a la posición antes de E  
 }  
 System.out.print((char) is.read()); // E  
 System.out.print((char) is.read()); // O  
 System.out.print((char) is.read()); // N  
}

El fragmento de código imprimirá “LEOEON” si mark() es compatible, y “LEON” en caso contrario. Es una buena práctica organizar las operaciones read() de modo que el flujo termine en la misma posición, independientemente de si mark() es compatible o no.

¿Y qué hay del **valor 100 que pasamos al método mark()**? Este valor **se llama readLimit**. Le indica al flujo que esperamos **llamar a reset() después de leer como máximo 100 bytes**. Si el programa llama a reset() después de leer más de 100 bytes al llamar a mark(100), entonces **podría lanzar una excepción, dependiendo de la clase de flujo**.

##### skip()

Supongamos que tenemos una instancia de InputStream cuyos próximos valores son “TIGRES”. Considera el siguiente fragmento de código:

System.out.print((char) is.read()); // T  
is.skip(2); // Salta I y G  
is.read(); // Lee R pero no lo muestra  
System.out.print((char) is.read()); // E  
System.out.print((char) is.read()); // S

Este código imprimirá “TES” en tiempo de ejecución. Hemos saltado dos caracteres, I y G. También leímos R pero no lo almacenamos en ninguna parte, por lo que se comporta como si hubiéramos llamado a skip(1).

El **valor devuelto por skip() nos indica cuántos valores se omitieron realmente** . Por ejemplo, si estamos cerca del final del flujo y llamamos a skip(1000), el valor de retorno podría ser 20, lo que indica que **se alcanzó el final del flujo después de omitir** 20 valores. Usar el valor devuelto por skip() es importante si necesitas llevar un registro de dónde estás en un flujo y cuántos bytes se han procesado.

#### 1.5. Flushing de flujos de salida (Output Streams)

Cuando se escribe datos en un flujo de salida, el sistema operativo subyacente **no garantiza que los datos se escriban inmediatamente en el sistema de archivos**. En muchos sistemas operativos, los **datos pueden almacenarse en la memoria**, y la escritura se produce solo después de que se llena una caché temporal o después de un cierto período de tiempo.

**Si los datos se almacenan en la memoria y la aplicación termina de manera inesperada, los datos se perderán**, ya que nunca se escribieron en el sistema de archivos. Para abordar esto, **todas las clases de flujos de salida proporcionan un método flush(), que solicita que todos los datos acumulados se escriban de inmediato en el disco**.

// OutputStream y Writer  
  
public void flush() throws IOException

En el siguiente ejemplo, se escriben 1000 caracteres en un flujo de archivo. Las llamadas a **flush() aseguran que los datos se envíen al disco** duro al menos una vez cada 100 caracteres. La JVM o el sistema operativo son libres de enviar los datos con más frecuencia.

try (var fos = new FileOutputStream(fileName)) {  
 for (int i = 0; i < 1000; i++) {  
 fos.write('a');  
 if (i % 100 == 0) {  
 fos.flush();  
 }  
 }  
}

El método **flush() ayuda a reducir la cantidad de datos perdidos si la aplicación termina de manera inesperada**. Sin embargo, no es gratuito. Cada vez que se usa, **puede causar un retraso perceptible en la aplicación, especialmente para archivos grandes**. A menos que los datos que estés escribiendo sean extremadamente críticos, el método **flush() solo debe usarse de manera intermitente**. Por ejemplo, no es necesario llamarlo después de cada escritura.

Tampoco es necesario llamar al método flush() cuando hayas terminado de escribir datos, ya que **==el método close() lo hará automáticamente=0**.

### 2. Resumen de métodos más comunes de flujos de E/S

La **Tabla 3** revisa los **métodos comunes de flujos que debes conocer para este apartado**.

Para los métodos read() y write() que toman arrays primitivos, el tipo de parámetro del método depende del tipo de flujo. Los flujos de bytes que terminan en InputStream/OutputStream utilizan byte[], mientras que los flujos de caracteres que terminan en Reader/Writer utilizan char[].

**Tabla 3:** Métodos de flujos de E/S más importantes

|  |  |  |
| --- | --- | --- |
| **Flujo** | **Nombre del Método** | **Descripción** |
| Todos los flujos | void close() | Cierra el flujo y libera los recursos |
| Todos los flujos de entrada | int read() | Lee un solo byte o devuelve -1 si no hay bytes disponibles |
| InputStream | int read(byte[] b) | Lee valores en un búfer. Devuelve el número de bytes leídos |
| Reader | int read(char[] c) | Lee valores en un búfer. Devuelve el número de bytes leídos |
| InputStream | int read(byte[] b, int offset, int length) | Lee hasta length valores en un búfer, comenzando desde la posición offset. Devuelve el número de bytes leídos |
| Reader | int read(char[] c, int offset, int length) | ee hasta length valores en un búfer, comenzando desde la posición offset. Devuelve el número de bytes leídos |
| Todos los flujos de salida | void write(int) | Escribe un solo byte |
| OutputStream | void write (byte[] b) | Escribe un array de valores en el flujo |
| Writer | void write(char[] c) | Escribe un array de valores en el flujo |
| OutpuStream | void write(byte[] c, int offset, int length) | Escribe length valores del array en un flujo, empezando desde el índice offset |
| Writer | void write(char[] c, int offset, int length) | Escribe length valores del array en un flujo, empezando desde el índice offset- |
| Todos los flujos de entrada | boolean markSupported() | Devuelve true si la clase de flujo admite mark() |
| Todos los flujos de entrada | void mark(int readLimit) | Marca la posición actual en el flujo |
| Todos los flujos de entrada | void reset() | Intenta restablecer el flujo a la posición marcada |
| Todos los flujos de entrada | long skip(long n) | Lee y descarta un número especificado de caracteres |
| Todos los flujos de salida | void flush() | Vacía los datos acumulados a través del flujo |

## EJERCICIOS

* [Boletín 01. Ejercicos con la clase File Y RandomAccessFile](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0109ejercicios/#bolet%C3%ADn-01-ejercicos-con-la-clase-file-y-randomaccessfile)
* [Boletín 02. Ejercicios con flujos I/O](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/01accesoficheros/01javaio/0109ejercicios/#bolet%C3%ADn-02-ejercicios-con-flujos-io)

### Boletín 01. Ejercicos con la clase File Y RandomAccessFile

Recuerda

Para realizar los ejercicios de este boletín, **debes crear un nuevo proyecto** en tu IDE preferido y **añadir las clases** que se indican en cada ejercicio. También debes consultar la [documentación oficial de la clase File](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/io/File.html) para conocer los métodos que puedes utilizar, así como el apartado: de “**Ventanas de entrada de datos, mensajes y archivos**” de la unidad de “**Refuerzo y ayudas complementarias**”, apartado “**Java General**”

Ejercicio 1. Creación y lectura de archivos con File

*Debes* ***trabajar únicamente con métodos de la clase File****.*

Realiza los siguientes pasos:

1. **Crea un archivo de texto** llamado prueba.txt en el directorio actual de tu proyecto, sólo si no existe.
2. **Escribe un programa** que cree un objeto File para el archivo prueba.txt y **compruebe si el archivo existe**.
3. **Si el archivo existe**, muestra la **ruta absoluta**, **nombre del archivo**, **tamaño**, **última modificación** y **si es un directorio**.
4. **Si el archivo no existe**, muestra un mensaje que lo indique y crea uno temporal.

Ejercicio 2. Mostrar el contenido de un directorio

*Debes* ***trabajar únicamente con métodos de la clase File****.*

El programa abre una ventana para la selección de un directorio (hazlo también desde teclado si recoge un parámetro) y usando el **método listFiles()** de la clase File, **muestra el contenido de ese directorio**, indicando el tamaño de los archivos y si es un directorio o no. Además, muestra el tamaño total de los archivos y directorios.

Muestra en una ventana emergente el resultado y por consola.

A continuación puedes ver algunas soluciones parciales del ejercicio 2. Completa el ejercicio de acuerdo a las indicaciones.

Solución parcial con list()

Solución parcial con JFileChooser

Solución completa con JFileChooser

Ejercicio 3. Gestor de archivos y directorios

Como en todos los ejercicios anteriores, debes **trabajar únicamente con métodos de la clase File**.

Escribe un programa en Java que funcione como un **gestor básico de archivos y directorios**. El programa debe permitir al usuario realizar las siguientes operaciones:

1. **Crear** un directorio, empleando la clase JFileChooser para seleccionar la ruta donde se creará.
2. **Listar** todos los archivos y subdirectorios de un directorio **de forma recursiva**.
3. **Eliminar** un archivo o directorio. Si es un directorio, eliminar todo su contenido de forma recursiva.
4. **Mover o renombrar** archivos y directorios.

El programa debe ofrecer un menú para que el usuario elija la operación que desea realizar. La selección de directorios o archivos debe realizarse con la clase JFileChooser.

Ejercicio 4. Escritura y lectura de archivos con RandomAccessFile

Escribe un programa que **escriba y lea datos en un archivo** usando la clase RandomAccessFile.

1. **Crea un archivo de texto** llamado prueba.txt en el directorio actual de tu proyecto, sólo si no existe.
2. **Escribe un programa** que cree un objeto RandomAccessFile para el archivo prueba.txt y **escriba un mensaje**.
3. **Lee el mensaje** y **muéstralo por consola**.

Ejercicio 5. Escritura y lectura de archivos con RandomAccessFile

Escribe un programa que utilice la clase RandomAccessFile para escribir en un archivo los números del 1 al 10 y luego los lea desde el archivo. Muestra los números leídos en la consola.

Solución al ejercicio 5

Ejercicio 6. Modificación de Contenido en un Archivo Binario con `RandomAccessFile`

Escribe un programa en Java que haga lo siguiente:

* Escriba 10 enteros en un archivo llamado “datos.bin”.
* Permita al usuario modificar el tercer número almacenado en el archivo por otro número.
* Muestra los números antes y después de la modificación en la consola.

Solución al ejercicio 6

Ejercicio 7. Escritura y lectura de archivos con RandomAccessFile

Escribe un programa que **escriba y lea datos en un archivo** usando la clase RandomAccessFile. El programa debe hacer lo siguiente:

1. **Crea un archivo de texto** llamado prueba.txt en el directorio actual de tu proyecto, sólo si no existe.
2. **Escribe un programa** que cree un objeto RandomAccessFile para el archivo prueba.txt y **escriba un mensaje**.
3. **Lee el mensaje** y **muéstralo por consola**.

### Boletín 02. Ejercicios con flujos I/O

Ejercicio 1. Copia de archivos I/O

Se debe realizar un programa para copiar archivos. El programa debe recoger el nombre del archivo origen y destino. Se existe debe solicitar confirmación sobrescribir.

Úsese I/O con buffer y métodos estáticos (tenga en cuenta que los archivos pueden ser binarios).

a) Para la lectura desde teclado puede emplearse la clase Scanner.

b) Realiza el mismo ejercicio, pero empleando entradas desde ventana con JFileChooser`` y mensajes de error en JOptionPane, si los hay.

c) Realiza un programa que lea con un JOptionPane pida una URL y para posteriormente abrir un JFileChooser para guardarlo en el disco local.

*Ayuda: para abrir un flujo de entrada a una URL puede hacerse con el método openStream() de URL. Ten en cuenta que puede lanzar excepciones:*

InputStream in = new URL(FILE\_URL).openStream();

d) Mejora el aparado a) para que la lectura de los datos lo haga en bloques (buffer) y no byte a byte.

Ejercicio 2. Serialización

Crea una clase Persona con los atributos nombre y edad. Genera un programa que serialice y deserialice un objeto de tipo Persona.

Debe tener un menú con las siguientes opciones:

1. Añadir persona.
2. Mostrar personas.
3. Buscar persona (por número o por nombre, según consideres)
4. Salir

Puedes hacerlo desde consola o por medio de una interfaz gráfica, haciendo uso de JOptionPane para introducir los datos ([JOptionPane.showInputDialog](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/swing/JOptionPane.html#showInputDialog(java.lang.Object))) y mostrar los resultados ([JOptionPane.showMessageDialog](https://docs.oracle.com/en/java/javase/23/docs/api/java.desktop/javax/swing/JOptionPane.html#showMessageDialog(java.awt.Component,java.lang.Object))).

Ejercicio 3. Serialización de colecciones

Crea una clase ColeccionPersonas que contenga una colección de objetos de tipo Persona. Implementa la interface Serializable y crea un programa que serialice y deserialice un objeto de tipo ColeccionPersonas.

Ejercicio 4. Lectura de URL

Crea un programa que lea el contenido de una URL y lo muestre por pantalla.

Mejore el programa para que p**ida una URL y la guarde en un archivo en una carpeta seleccionada del disco mediante un JFileChooser.**

¿Serías capaz de **mostrar el tamaño del contenido de la URL**? ¿Y que ponga la **extensión adecuada al archivo**?

Ayuda: Puedes emplear HttpURLConnection para obtener el tamaño del contenido y para obtener el Content-Type puedes emplear el método getContentType().

Ejercicio 5. Lectura de URL con HttpURLConnection

Amplía el ejercicio anterior para que emplee HttpURLConnection y **muestre la información de la cabecera HTTP**.

Ejercicio 6. Estadísticas de un archivo

Realice un programa que **recoja el nombre de un fichero y muestre una estadística de la ruta, número de líneas, número de espacios, número de letras, fecha última modificación, longitud del fichero, …** Defina una clase **EstatisticaFile** con atributos: letras, linhas, espacios, archivo (tipo [File](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/File.html)).

private File arquivo;  
private int linhas;  
private int letras;  
private int espazos;

Métodos para obtener cada uno de los atributos, existe(), ultimaModificacion(), getRuta(). El constructor recoge el nombre del archivo.

Ejercicio 7. Estadísticas de un archivo con RandomAccessFile

Realice un programa que recoja el nombre de un fichero y muestre una estadística de la ruta, número de líneas, número de espacios, número de letras, fecha última modificación, longitud del fichero, …

Defina una clase **EstatisticaFile** con atributos: letras, linhas, espacios, archivo (tipo [File](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/io/File.html)).

private File arquivo;  
private int linhas;  
private int letras;  
private int espazos;

Métodos para obtener cada uno de los atributos, existe(), ultimaModificacion(), getRuta(). El constructor recoge el nombre del archivo.

Ejercicio 8. Editor de texto

Haz un programa que recoja el nombre de un fichero y muestre su contenido si existe o cree un nuevo en el que puedas escribir si no existe. Ejemplo: java Editor proba.txt

Para tal fin, además del programa, **Editor.java, crea la clase Documento** con las siguientes características:

1. Propiedades: arquivo (de tipo File)
2. Constructores: **recoge el nombre del archivo y crea el objeto archivo**. Otro que recoja un Objeto de tipo File.
3. Métodos:
4. **exists()**: devuelve verdadero cuando el fichero no es nulo y existe.
5. **readFile()**: devuelve una cadena con el contenido del archivo, si existe, obviamente. Emplea StringBuilder.
6. **readFileNIO()**: igual al anterior, pero empleado Path y el método readString de Files.
7. **writeFromString(…)**: recoge una cadena y la escribe en fichero, al final, empleando BufferedWriter.
8. **writeFromStringPrintWriter(…**): recoge una cadena y la escribe al final, empleando PrintWriter.
9. **writeFromInputStream()**: rue recoge un flujo de tipo InputStream (para, por ejemplo, System.in) y escribe lo recogido por el flujo en el fichero.
10. **writeFromKeyword()**: escribe en el archivo lo que se escriba en el teclado.
11. . **getFile()**: devuelve el objeto archivo.
12. **toString()**: devuelve la ruta absoluta/canónica al archivo.

**AppEditor.java** recoge el nombre por línea de órdenes. Si existe, muestra el contenido (llama al método readFile()) si no existe pide que introduzcas por teclado. Para acabar de introducir datos debe escribir una línea que sólo contiene un “.”.

Ejercicio 9.Lectura de teclado

Realiza una clase de utilidad **Teclado** c**on métodos y atributos estáticos para leer desde teclado**, que tenga un **atributo estático privado LECTOR de tipo BufferedReader** (lector de caracteres con buffer que permite leer línea la línea). La clase debe ter los siguientes métodos: lerString, lerChar, lerInt, lerLong, lerBoolean, lerFloat, lerDouble, lerByte, lerShort, para cada tipo de dato básico. Haz un pequeño programa que haga uso esta clase.

Ayuda: emplead el atributo estático System.in (de tipo java.io.InputStream), así como la clase correspondiente que permita pasar un flujo de tipo Byte a un flujo de tipo Carácter.

Como sabéis, J**ava ya incorpora clases para facilitar la lectura desde teclado**: java.io.Console (java 1.6 y sup.) e java.util.Scanner (java 1.5 e sup.), entre otras, como BufferedReader (java 1.1 y sup.).

Ejercicio 10. Gestión de equipos de fútbol

Haga un programa de gestión de la clasificación de la liga de fútbol. Declare una clase **Equipo** con los atributos mínimos necesarios: nome, ganhados, perdidos, empatados, golesFavor, golesContra.

Para poder ordenar los equipos **debe implantar a interface** [**Comparable**](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Comparable.html), y para poder guardarse con el método writeObject de ObjectOutputStream debe implantar Serializable.

Sobrescribe **el método equals para que dos Equipos sean iguales si tienen el mismo nombre** (¡¡¡¡**implanta hashCode()!!!!**)

Los equipos deben guardarse en un fichero “**clasificacion.dat**”. El programa debe tener un menú con las siguientes opciones: cargar equipos, añadir equipo, guardar equipos, mostrar clasificación, modificar equipo.

Una vez cargados emplee un objeto de tipo TreeSet para que los ordene correctamente.

Ejercicio 11. Gestión de equipos de baloncesto

Haga un programa para la **gestión y clasificación de la liga de baloncesto**. La clasificación de los equipos se **guarda en un archivo llamado clasificacion.dat**.

a) Declare una **clase Equipo** con los atributos mínimos necesarios: nombre, victorias, derrotas, puntosAfavor a favor, puntosEnContra puntos en contra. Puedes añadir los atributos que te interesen, como ciudad, etc. Tienes libertad para hacerlo, pues, además, te puede servir como práctica.

Tenga en cuenta que los atributos **puntos, partidos jugados y diferencia de puntos son atributos derivados** que se calculan a partir de los partidos ganados, perdidos, puntos a favor y puntos en contra.

Cree los métodos que considere oportunos, pero tome decisiones sobre los métodos get/set necesarios. Así, haz un método que devuelva los puntos, getPuntos, un método getPartidosJugados que devuelva el número de partidos jugados y un método getDiferenciaDePuntos, que devuelva la diferencia de puntos. Obviamente, por ser atributos/propiedades derivados/as, no tienen sentido los métodos de tipo “set” para ellos.

Debe tener, al menos, un constructor para la clase equipo que recoja el nombre y otro que recoja todas las propiedades. No debe existir un constructor por defecto (en la práctica sí si debería tener).

Para poder ordenar los equipos d**ebe implantar la interface Comparable<Equipo>**. Piense que debe ordenar por puntos y, a igualdad de puntos, por diferencia de puntos encestados. Además, para poder guardar los objetos (writeObject de ObjectOutputStream) y/o recuperarlos (readObject de ObjectInputStream) **debe implantar la interface Serializable**. Lo mismo con la clase siguiente, Clasificacion, que debe implementar la interface Serializable.

**Sobrescribe el método equals** para que se considere que dos Equipos son iguales si tienen el mismo nombre (sin distinguir mayúsculas de minúsculas). Haz lo mismo con hashCode.

b) Declare una **clase Clasificacion**, con un atributo **equipos de tipo ArrayList** de Equipo, aunque debe existir un constructor que permita crear una clasificación con los equipos que se desee. Defina los métodos para añadir equipos a la clasificación, addEquipo, así como los métodos para eliminar equipo, removeEquipo, y sobrescriba el método toString que devuelva la cadena de la clasificación (StringBuilder)

Crea los métodos estáticos: \*\*loadClasificacion\*\*, que cargue la clasificación del archivo y la devuelva, y el método \*\*saveClasificacion\*\*, que guarde la clasificación en el archivo.

Una vez cargados se podría emplear un objeto de tipo ***TreeSet*** para que ordene correctamente la clasificación (lo veremos en unidades posteriores)

c) El programa debe tener un menú con las siguientes opciones: a. añadir equipo (pide el nombre del equipo y los valores de los atributos no derivados, añadiendo el equipo a la clasificación) b. mostrar clasificación (muestra la clasificación ordenada de los equipos que están cargados en memoria) c. guardar clasificación (que guarda la clasificación en el archivo clasificacion.dat) d. cargar clasificación (que carga la clasificación del archivo clasificacion.dat) e. salir (sale del programa, debiendo preguntar antes).

Utilice la clase Scanner para leer de teclado.

Ejercicio 12. Lectura de un archivo BMP

**Modificación de un archivo BMP**.

1. Haga un programa que **lea la cabecera de un archivo BMP sin compresión de 24 bits** (un archivo de 24 bits implica que cada pixel se representa con 3 bytes, uno para cada color RGB) y muestre la información de la cabecera. Emplee un **flujo de tipo DataInputStream**.

La clase DataInputStream permite leer datos primitivos de un flujo de entrada en un formato de datos binarios. Cada método de esta clase lee un dato primitivo de un flujo de entrada en un formato de datos binarios adecuado y devuelve el valor correspondiente.

Para leer los bytes de la cabecera, emplee el método readByte() de la clase DataInputStream o puedes leer todos los bytes con el método readFully(byte[] b).

Puedes emplear el método toBinaryString de la clase Integer para mostrar los bytes en binario.

Defina una clase **Cabecera** que recoja el nombre del archivo y tenga los atributos necesarios para guardar la información del mismo.

/\*  
 \* 2 signature, must be 4D42 hex  
 \* 4 size of BMP file in bytes (unreliable)  
 \* 2 reserved, must be zero  
 \* 2 reserved, must be zero  
 \* 4 offset to start of image data in bytes  
 \* 4 size of BITMAPINFOHEADER structure, must be 40  
 \* 4 image width in pixels  
 \* 4 image height in pixels  
 \* 2 number of planes in the image, must be 1  
 \* 2 number of bits per pixel (1, 4, 8, or 24)  
 \* 4 compression type (0=none, 1=RLE-8, 2=RLE-4)  
 \* 4 size of image data in bytes (including padding)  
 \* 4 horizontal resolution in pixels per meter (unreliable)  
 \* 4 vertical resolution in pixels per meter (unreliable)  
 \* 4 number of colors in image, or zero  
 \* 4 number of important colors, or zero  
 \*/

Ayuda: para pasar el array de 4 bytes a un entero, puede emplear el método ByteBuffer.wrap(byte[]).order(ByteOrder.LITTLE\_ENDIAN).getInt(). En el caso de dos bytes puedes emplear ByteBuffer.wrap(byte[]).order(ByteOrder.LITTLE\_ENDIAN).getShort().

También puedes hacer uso del siguiente método, que trabaja a más bajo nivel:

public static int byteAInt(byte[] bytes) {  
 return ((bytes[3] & 0xFF) << 24) | ((bytes[2] & 0xFF) << 16) | ((bytes[1] & 0xFF) << 8) | (bytes[0] & 0xFF);  
}

O, para short:

public static int byteAInt(byte[] bytes) {  
 return ((bytes[1] & 0xFF) << 8) | (bytes[0] & 0xFF);  
}

La **máscara es necesaria porque Java no tiene tipos sin signo y al hacer el cast a int, los bytes se convierten a enteros con signo**. Por ejemplo: si el byte es 0xFF (255), al convertirlo a entero, se convierte en -1. **El operador desplazamiento a la izquierda (<<) desplaza los bits a la izquierda y rellena con ceros a la derecha**. **Si el tipo de dato es byte, se convierte a int antes de hacer la operación**.

1. Diseña e implanta de un programa que **lea la cabecera de un BMP y permita invertir la imagen, pasarla a escala de grises, añadir ruido, aclarar y oscurecer**. La imagen está a continuación de la cabecera. Para pasar la escala de grises hay que establecer los 3 colores del píxel al mismo nivel con la media de los colores.

Solución parcial lectura archivo BMP

import java.io.DataInputStream;  
import java.io.FileInputStream;  
import java.io.IOException;  
  
public class LeerCabeceraBMP {  
 public static void main(String[] args) {  
 try (DataInputStream dis = new DataInputStream(new FileInputStream("imagen.bmp"))) {  
 byte[] cabecera = new byte[54];  
 dis.readFully(cabecera); // Guardamos la cabecera en un array de bytes  
 System.out.println("Cabecera BMP:");  
 System.out.println("Signature: " + new String(cabecera, 0, 2)); // La sinatura BMP es 4D42, de 2 bits  
 System.out.println("Size: " + byteAInt(cabecera, 2)); // Convierte los 4 bytes a un entero en formato LITTLE\_ENDIAN  
 System.out.println("Offset: " + byteAInt(cabecera, 10)); // Offset a los datos de la imagen  
 System.out.println("Width: " + byteAInt(cabecera, 18)); // Ancho de la imagen  
 System.out.println("Height: " + byteAInt(cabecera, 22)); // Alto de la imagen  
 System.out.println("Bits per pixel: " + byteAInt(cabecera, 28)); // Bits por pixel  
 } catch (IOException e) {  
 System.out.println("Error de entrada/salida: " + e.getMessage());  
 }  
 }  
  
 public static int byteAInt(byte[] bytes, int offset) {  
 return ((bytes[offset + 3] & 0xFF) << 24) | ((bytes[offset + 2] & 0xFF) << 16) | ((bytes[offset + 1] & 0xFF) << 8) | (bytes[offset] & 0xFF);  
 }  
}

Solución completa Cabecera archivo BMP

package com.pepinho.ad.e06bmp;  
  
/\*\*  
 \*  
 \* @author pepecalo  
 \*/  
  
/\*  
 \* 2 signature, must be 4D42 hex  
 \* 4 size of BMP file in bytes (unreliable)  
 \* 2 reserved, must be zero  
 \* 2 reserved, must be zero  
 \* 4 offset to start of image data in bytes  
 \* 4 size of BITMAPINFOHEADER structure, must be 40  
 \* 4 image width in pixels  
 \* 4 image height in pixels  
 \* 2 number of planes in the image, must be 1  
 \* 2 number of bits per pixel (1, 4, 8, or 24)  
 \* 4 compression type (0=none, 1=RLE-8, 2=RLE-4)  
 \* 4 size of image data in bytes (including padding)  
 \* 4 horizontal resolution in pixels per meter (unreliable)  
 \* 4 vertical resolution in pixels per meter (unreliable)  
 \* 4 number of colors in image, or zero  
 \* 4 number of important colors, or zero  
 \*/  
  
import java.io.\*;  
  
public class CabeceraBMP {  
  
 public static final String BARRA = "==========================================";  
 public static final String NL = System.lineSeparator();  
 public static final int TAMANHO = 54;  
  
 private byte[] cabeceraBytes;  
  
 public CabeceraBMP(String arquivo) {  
 this(new File(arquivo));  
 }  
  
 public CabeceraBMP(File f) {  
 this.cabeceraBytes = new byte[TAMANHO];  
 try ( DataInputStream dataInputStream = new DataInputStream(  
 new FileInputStream(f));) {  
 dataInputStream.readFully(cabeceraBytes);  
 } catch (FileNotFoundException ex) {  
 System.err.println(ex.getMessage());  
 } catch (IOException ex) {  
 System.err.println(ex.getMessage());  
 }  
 }  
  
 public String getSinature() {  
 return new String(cabeceraBytes, 0, 2);  
 }  
  
 public int getTamanoArquivo() {  
 return byteArrayToInt(cabeceraBytes, 2);  
 }  
  
 public int getReserva1() {  
 return byteArrayToShort(cabeceraBytes, 6);  
 }  
  
 public int getReserva2() {  
 return byteArrayToShort(cabeceraBytes, 8);  
 }  
  
 public int getOffsetImage() {  
 return byteArrayToInt(cabeceraBytes, 10);  
 }  
  
 public int getInfoHeader() {  
 return byteArrayToInt(cabeceraBytes, 14);  
 }  
  
 public int getAnchura() {  
 return byteArrayToInt(cabeceraBytes, 18);  
 }  
  
 public int getAltura() {  
 return byteArrayToInt(cabeceraBytes, 22);  
 }  
  
 public int getNumeroPlanos() {  
 return byteArrayToShort(cabeceraBytes, 26);  
 }  
  
 public int getBitsPerPixel() {  
 return byteArrayToShort(cabeceraBytes, 28);  
 }  
  
 public int getTipoCompresion() {  
 return byteArrayToInt(cabeceraBytes, 30);  
 }  
  
 public String getTipoCompresionAsString() {  
 int tipo = getTipoCompresion();  
 switch (tipo) {  
 case 0 -> {  
 return "Sin compresión";  
 }  
 case 1 -> {  
 return "RLE-8";  
 }  
 case 2 -> {  
 return "RLE-4";  
 }  
 default ->  
 throw new AssertionError();  
 }  
  
 }  
  
 public int getTamanoImagen() {  
 return byteArrayToInt(cabeceraBytes, 34);  
 }  
  
 public int getResolucionHorizontalPorMetro() {  
 return byteArrayToInt(cabeceraBytes, 38);  
 }  
  
 public int getResolucionVerticalPorMetro() {  
 return byteArrayToInt(cabeceraBytes, 42);  
 }  
  
 public int getNumeroColores() {  
 return byteArrayToInt(cabeceraBytes, 46);  
 }  
  
 public int getImportanciaColores() {  
 return byteArrayToInt(cabeceraBytes, 50);  
 }  
  
 // Método para convertir un array de bytes en un entero de 4 bytes (little endian)  
 public static int byteArrayToInt(byte[] bytes, int offset) {  
 return ((bytes[offset + 3] & 0xFF) << 24)  
 | ((bytes[offset + 2] & 0xFF) << 16)  
 | ((bytes[offset + 1] & 0xFF) << 8)  
 | (bytes[offset] & 0xFF);  
 }  
  
 public static int byteArrayToShort(byte[] bytes, int offset) {  
 return ((bytes[offset + 1] & 0xFF) << 8)  
 | (bytes[offset] & 0xFF);  
 }  
  
 /\*  
  
  
  
 \* 4 horizontal resolution in pixels per meter (unreliable)  
 \* 4 vertical resolution in pixels per meter (unreliable)  
 \* 4 number of colors in image, or zero  
 \* 4 number of important colors, or zero  
 \*/  
 @Override  
 public String toString() {  
 StringBuilder sb = new StringBuilder();  
 sb.append("Cabecera BMP:\n").append(BARRA)  
 .append("Firma: ").append(getSinature()).append(NL)  
 .append("Tamaño arquivo: ").append(getTamanoArquivo()).append(NL)  
 .append("Reserva 1: ").append(getReserva1()).append(NL)  
 .append("Reserva 2: ").append(getReserva2()).append(NL)  
 .append("Offset datos imagen: ").append(getOffsetImage()).append(NL)  
 .append("BITMAPINFOHEADER: ").append(getInfoHeader()).append(NL)  
 .append("Anchura: ").append(getAnchura()).append(" píxeles").append(NL)  
 .append("Altura: ").append(getAltura()).append(" píxeles").append(NL)  
 .append("Número de planos: ").append(getNumeroPlanos()).append(NL)  
 .append("Bits por píxel: ").append(getBitsPerPixel()).append(NL)  
 .append("Tipo de compresión: ").append(getTipoCompresionAsString()).append(NL)  
 .append("Tamaño de la imagen: ").append(getTamanoImagen()).append(" bytes").append(NL)  
 .append("Resolución horizontal: ").append(getResolucionHorizontalPorMetro()).append(NL)  
 .append("Resolución vertical: ").append(getResolucionVerticalPorMetro()).append(NL)  
 .append("Número de colores: ").append(getNumeroColores()).append(NL)  
 .append("Importancia de colores: ").append(getImportanciaColores()).append(NL);  
 return sb.toString();  
 }  
  
 public static void main(String[] args) {  
 // Ruta del archivo BMP a leer  
 String archivoBMP = "e:\\putin.bmp"; //   
  
 CabeceraBMP cabecera = new CabeceraBMP(archivoBMP);  
 System.out.println(cabecera);  
 }  
  
}

Solución pasar a escala de crises archivo BMP

import java.io.\*;  
  
public class EscalaGrisesBMP {  
  
 public static void main(String[] args) {  
 try (DataInputStream dis = new DataInputStream(new FileInputStream("e:\\putin.bmp"))) {  
 byte[] cabecera = new byte[54];  
 dis.readFully(cabecera); // Guardamos la cabecera en un array de bytes  
 int ancho = ((cabecera[21] & 0xFF) << 24) | ((cabecera[20] & 0xFF) << 16) | ((cabecera[19] & 0xFF) << 8) | (cabecera[18] & 0xFF);  
 int alto = ((cabecera[25] & 0xFF) << 24) | ((cabecera[24] & 0xFF) << 16) | ((cabecera[23] & 0xFF) << 8) | (cabecera[22] & 0xFF);  
 int bitsPorPixel = ((cabecera[29] & 0xFF) << 8) | (cabecera[28] & 0xFF);  
 int tamanoImagen = ((cabecera[37] & 0xFF) << 24) | ((cabecera[36] & 0xFF) << 16) | ((cabecera[35] & 0xFF) << 8) | (cabecera[34] & 0xFF);  
 byte[] imagen = new byte[tamanoImagen];  
 dis.readFully(imagen); // Guardamos la imagen en un array de bytes  
 byte[] imagenGrises = new byte[tamanoImagen];  
 for (int i = 0; i < tamanoImagen; i += 3) {  
 byte promedio = (byte) ((imagen[i] + imagen[i + 1] + imagen[i + 2]) / 3);  
 imagenGrises[i] = promedio;  
 imagenGrises[i + 1] = promedio;  
 imagenGrises[i + 2] = promedio;  
 }  
 try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("imagen\_grises.bmp"))) {  
 dos.write(cabecera);  
 dos.write(imagenGrises);  
 }  
 } catch (IOException e) {  
 System.out.println("Error de entrada/salida: " + e.getMessage());  
 }  
 }  
}

Solución añadir ruido archivo BMP

import java.io.\*;  
  
public class RuidoBMP {  
  
 public static void main(String[] args) {  
 try (DataInputStream dis = new DataInputStream(new FileInputStream("e:\\putin.bmp"))) {  
 byte[] cabecera = new byte[54];  
 dis.readFully(cabecera); // Guardamos la cabecera en un array de bytes  
 int ancho = ((cabecera[21] & 0xFF) << 24) | ((cabecera[20] & 0xFF) << 16) | ((cabecera[19] & 0xFF) << 8) | (cabecera[18] & 0xFF);  
 int alto = ((cabecera[25] & 0xFF) << 24) | ((cabecera[24] & 0xFF) << 16) | ((cabecera[23] & 0xFF) << 8) | (cabecera[22] & 0xFF);  
 int bitsPorPixel = ((cabecera[29] & 0xFF) << 8) | (cabecera[28] & 0xFF);  
 int tamanoImagen = ((cabecera[37] & 0xFF) << 24) | ((cabecera[36] & 0xFF) << 16) | ((cabecera[35] & 0xFF) << 8) | (cabecera[34] & 0xFF);  
 byte[] imagen = new byte[tamanoImagen];  
 dis.readFully(imagen); // Guardamos la imagen en un array de bytes  
 byte[] imagenRuido = new byte[tamanoImagen];  
 for (int i = 0; i < tamanoImagen; i++) {  
 imagenRuido[i] = (byte) (imagen[i] + (Math.random() \* 255 - 128));  
 }  
 try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("imagen\_ruido.bmp"))) {  
 dos.write(cabecera);  
 dos.write(imagenRuido);  
 }  
 } catch (IOException e) {  
 System.out.println("Error de entrada/salida: " + e.getMessage());  
 }  
 }  
}