

# Tema 1

## Manejo de ficheros.

### 1.1. Introducción.

Información: componente más importante de un sistema informático y el más difícil de recuperar ante un problema.

#### 1.1.1. Persistencia.

Datos que se guardan en ficheros → datos persistentes: persisten más allá de la ejecución de la aplicación que los trata.

Gestionar la persistencia de datos:

- Ficheros.
- Bases de datos.
- Mapeo objeto relacional (ORM): convertir datos entre un lenguaje de POO y una BdD relacional. Ej: Hibernate, Django ORM, SQL Alchemy, Microsoft Entity Framework...
- Bases de datos XML:
  - Bases de datos nativas XML: El SGBD almacena y recupera los datos en archivos XML. Ej: eXist, Apache Xindice, Berkeley dbXML....
  - BdD con capacidad XML (XML enabled): SGBD relacionales que pueden trabajar con XML. Ej: IBM DB2 (pureXML), Microsoft SQL Server, Oracle Database, PostgreSQL...

#### 1.1.2. Repaso de ficheros.

Tipos:

- De texto. Codificaciones:
  - UTF-8
  - ISO 8859-1
  - UTF-16
- Binarios.

Conceptos:

- Archivo o fichero.
- Carpeta o directorio.
- Extensión del fichero.
- Rutas:
  - Relativas.
  - Absolutas.

## 1.2. Propiedades del sistema.

Para poder automatizar el trabajo de ficheros independizándolo del sistema, conviene conocer [System Properties](#).

Propiedades interesantes:

- `"file.separator"`
- `"user.home"`
- `"user.dir"`
- `"line.separator"`

### Cuestión:

Consultando el enlace anterior, determina qué hace cada una de las propiedades anteriores.

Para consultar la propiedad se usa `System.getProperty(String)`:

```
separador = System.getProperty("file.separator");

System.out.println("Caracter para separar archivos del sistema: " +
    separador);
```

### 1.3. La clase `java.io.File`.

La librería `java.io` contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java.

La clase `java.io.File` es el mecanismo basado en *Streams* utilizado para E/S de ficheros a partir de Java 1.4.

**IMPORTANTE:** Las instancias de la clase `File` representan la ruta y el nombre del archivo, no el archivo en sí mismo.

#### Cuestión:

En la clase `File` existe un campo que contiene el mismo valor que nos devolvía `System.getProperty("file.separator")`, ¿cuál es?

Hay diferentes formas de crear el archivo. En los ejemplos usaré rutas de Windows pero dependerá del sistema.

```
// Path completo barra invertida escapada.
File fichero1 = new File("C:\\AccesoADatos\\fich1.txt");

// Path en partes (directorio, archivo).
File fichero2 = new File("C:/AccesoADatos", "fich2.txt");

// Directorio en instancia de File.
File directorio1 = new File("C:\\AccesoADatos");
File fichero3 = new File(directorio1, "fich3.txt");
```

#### 1.3.1. Métodos más importantes.

- Creación:

- `createNewFile()`: crea un nuevo fichero, vacío, asociado a `File` si y sólo si no existe un fichero con dicho nombre.
- `mkdir()`: crea un directorio con el nombre indicado en la creación del objeto `File`.
- `mkdirs()`: crea también los directorios superiores si no existen.
- Información:
  - `exists()`: devuelve `true` si el fichero/directorio existe.
  - `getName()`: devuelve el nombre del fichero o directorio.
  - `getPath()`: devuelve el camino relativo.
  - `getAbsolutePath()`: devuelve el camino absoluto del fichero/directorio.
  - `canRead()`: devuelve `true` si el fichero se puede leer.
  - `canWrite()`: devuelve `true` si el fichero se puede escribir.
  - `length()`: nos devuelve el tamaño del fichero en bytes.
  - `getParent()`: devuelve el nombre del directorio padre, o `null` si no existe.
  - `isDirectory()`: devuelve `true` si el objeto `File` corresponde a un directorio.
  - `isFile()`: devuelve `true` si el objeto `File` corresponde a un fichero normal.
  - `list()`: si el objeto `File` es un directorio, devuelve un array con un listado de `Strings` con los nombres de los archivos y directorios que contiene.
  - `listFiles()`: igual que el anterior pero el array contendrá un listado de objetos `File`.
- Operaciones sobre el fichero:
  - `delete()`: borra el fichero o directorio asociado al `File`.
  - `deleteOnExit()`: se borra cuando finaliza la ejecución de la máquina virtual Java.
  - `renameTo(File "nuevoNombre")`: renombra el fichero. El objeto `File` dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.

### 1.3.2. Unificar separador en las rutas.

A la hora de implementar métodos que por ejemplo cojan nombres de directorios de una ruta, necesitaremos el separador que dependerá del sistema operativo.

Una solución es usar `System.getProperty("file.separator")` o `File.separator` cada vez que lo necesitemos. Muchas veces será lo mejor pero otras, como por ejemplo a la hora de construir expresiones regulares, sería mejor que las rutas tengan el mismo separador independientemente del sistema en el que estemos.

**Nota:**

El uso de letras de unidad en Windows y un único origen en UNIX no se puede unificar.

El siguiente método transformaría la ruta.

```
/**
 * Transforma las barras de una ruta Windows.
 * @param ruta La ruta a transformar.
 * @return La ruta transformada si es correcta.
 *
 * Por evitar dejar un catch sin código se ha introducido una
 * asignación de la expresión a null pero nunca debería
 * producirse ya que la expresión se a escrito a mano en el
 * método.
 */
public static String substSepArchivos(String ruta) {
    String separadorWin = "\\";
    /* Parece complicado pero en expresiones regulares, \ indica
    caracter escapado, por lo que \\ sería \
    Como en Strings de Java \ también indica caracter escapado, \\ \\
    sería el String \\ que como expresión regular indica \
    */
    String separadorWinExp = "\\\\";
    String separadorUnix = "/";
    String rutaModificada = null;

    try {
        // Comprobamos si es ruta Windows
        System.out.println(File.separator.equals(separadorWin));
        if ( File.separator.equals(separadorWin) ) {
            rutaModificada = ruta.replaceAll(separadorWinExp,
                separadorUnix);
        }
    }
    catch(java.util.regex.PatternSyntaxException e) {
        rutaModificada = null;
    }

    return rutaModificada;
}
```

**Debate:**

Estudia el ejemplo anterior y discútelo en clase.

**Nota:**

Generalmente no incluiré Javadoc en el código para hacerlo más corto.

**Ejercicios 1.1**

Siempre que lo necesites consulta la documentación de la [API](#) de Java.

1. Crea un método `listarDirectorio()` que devuelva una array con el listado del contenido (archivos y carpetas) del directorio actual.  
¿Este método debería ser dinámico o estático? ¿por qué?
2. Crea un método `listarDirectorio(String directorio)` que devuelva una array con el listado del contenido del directorio indicado como argumento siempre y cuando este sea un directorio y no un archivo. Pruébalo pasándole al menos una ruta absoluta y una relativa.  
¿Qué devolvería en caso de que la ruta que nos proporcionan no se correspondiera con un directorio?
3. Crea un método `existeFichero(String directorio, String fichero)` que compruebe si existe dicho fichero en el directorio indicado.
4. Crea un método `generarArchivo` que a partir de una ruta que se le pase como argumento, cree un archivo txt con nombre `TunombreTuapellido` en la ruta en la que se le ha proporcionado. Presta atención a los posibles errores que puedan darse.  
¿Qué pasa si la ruta no existe? ¿puedes solucionarlo?
5. Crea un método `renombrarArchivo` que coja un archivo cuyo path absoluto se le pase como argumento y lo renombre añadiendo delante `DAM2`.  
Pruébalo con el archivo creado en el ejercicio anterior.  
El archivo antiguo, ¿desaparece?
6. Crea un método que se llame `borrarArchivo` que reciba un path absoluto y elimine el archivo indicado.  
Pruébalo con el archivo del ejercicio anterior.  
En la clase `File` hay métodos para cambiar los atributos del archivo. Prueba a modificar el método haciendo el archivo de solo lectura antes de eliminarlo. ¿Qué sucede? ¿por qué?
7. Crea un método `eliminarDirectorio` que reciba una ruta y elimine el directorio indicado por ella.  
¿Elimina directorios con contenido? ¿cómo se puede solucionar?

Modifica el método para que elimine directorios que contengan solo archivos o estén vacíos e indique que no puede hacerlo si contienen otros directorios.

8. Crea una clase que herede de `java.io.File` y le añada un método que se llame `showInfo()`. Este método devolverá un `String` con la siguiente información del fichero:

- Nombre.
- Ruta.
- Ruta absoluta.
- ¿Se puede leer?
- ¿Se puede escribir?
- Tamaño.
- ¿Es un directorio? En caso afirmativo mostrará los contenidos del mismo.
- ¿Es un fichero?

Cada elemento debe ir en una línea y llevar delante un texto que indique a qué se refiere.

---

### 1.3.3. Problemas.

Problemas de la clase `java.io.File`:

- Limitada en cuanto a proporcionar detalles sobre errores específicos. Por lo general, solo te dirá que algo falló, pero no por qué falló. Por ejemplo si hay un fallo al borrar un fichero no es posible saber si es porque no existe, no se tienen los permisos adecuados, etc.
- Muchos métodos no lanzan excepciones cuando fallan.
- El funcionamiento en diferentes plataformas es poco consistente. Dificulta la portabilidad.
- El soporte para enlaces simbólicos es limitado.
- El manejo de los permisos, propiedades y atributos de seguridad debería ser más completo.
- Los métodos no están preparados para trabajar con grandes cantidades de datos: listar un directorio muy grande puede fallar y causar pérdidas de información en memoria.
- Faltan operaciones básicas como copiar, mover, manejar enlaces simbólicos, etc.
- Los métodos son bloqueantes, lo que significa que el hilo que realiza la operación de I/O se bloquea hasta que la operación esté completa. Esto puede ser



ineficiente en aplicaciones que necesitan realizar muchas operaciones de I/O.

- no ofrece muchas operaciones atómicas. Por ejemplo, si deseas mover un archivo y sobrescribirlo si ya existe, tendrás que realizar múltiples operaciones separadas, lo que puede llevar a problemas de concurrencia.

## 1.4. El paquete `java.nio.file`

El paquete `java.nio`<sup>1</sup> se introduce en Java 1.4. Es recomendable usarlo en lugar de `java.io.File`.

IO vs NIO:

- Basado en `Buffers` en lugar de `Streams`.
- Proporciona un mejor manejo de errores a través de excepciones más específicas.
- NIO no es bloqueante (permite operaciones asíncronas). No se queda esperando a tener algo que leer o a terminar una operación.
- Muchas operaciones, como mover o copiar archivos, pueden hacerse de manera atómica, lo que mejora la integridad de los datos.
- Proporciona un conjunto más completo de funcionalidades para manipular archivos y directorios, incluida la capacidad para trabajar con enlaces simbólicos, permisos de archivo, etc.
- NIO introduce el concepto de `Channel` que proporciona un método eficiente para transmitir la información entre el emisor y el buffer. Un canal:
  - Lee información en un buffer y la proporciona a un elemento de E/S (p.ej: fichero).
  - Escribe información desde un buffer desde un elemento de E/S.
- NIO introduce el concepto de `Selector` que permite elegir entre los canales disponibles para operaciones de E/S sin necesidad de usar varios hilos<sup>2</sup>. Un hilo puede monitorizar múltiples canales de entrada o de salida con un solo selector y será notificado cuando ocurra algún evento en ellos.
- Walk File Tree: ofrece métodos para recorrer fácilmente un árbol de archivos, lo que es más difícil de implementar de forma eficiente con `java.io.File`.
- Puedes usar `Files.newDirectoryStream` con un filtro para listar archivos que cumplan con ciertas condiciones, sin tener que listar todos los archivos primero como era necesario con `java.io.File`.

---

<sup>1</sup>NIO: New Input/Output

<sup>2</sup>threads

Elegir entre ellos:

- Pueden ser complementarios y utilizaremos conjuntamente ambos paquetes.
- `java.nio` permite manejar múltiples canales (archivos o conexiones de red) con uno o unos pocos hilos.
- En `java.nio` el procesamiento de datos es más complicado que usar los *streams* bloqueantes de `java.io`.
- `java.nio` es mejor opción si necesito manejar muchas conexiones (canales) abiertas y en cada una manejar una pequeña cantidad de datos. (chats, Red P2P ...)
- `java.io` es mejor opción si voy a manejar pocas conexiones con mucha información cada vez.

#### IO vs. NIO vs. NIO2:

- IO: paquete antiguo. Bloqueante.
- NIO: introducido en Java 1.4. No bloqueante.
- NIO2: añade funcionalidades a NIO. Principalmente relacionadas con manejos de ficheros y sistemas de ficheros. Casi toda la nueva funcionalidad se concentra en `java.nio.file` aunque también añade algunos elementos al paquete superior `java.nio`.

Nos interesa `java.nio.file`. De todos los elementos que contiene los más interesantes son:

- **Files**: contiene métodos estáticos para operar con ficheros y directorios.
- **Path**: interfaz que representa un objeto que permite localizar un archivo en el sistema de archivos.
- **Paths**: contiene métodos estáticos para generar un `Path` desde un `String` o un `URI`.

### 1.4.1. Trabajar con Path y Paths.

**Nota:**

Tanto `java.io.File` como `java.nio.Path` tienen métodos para convertirse en el otro: `toPath` y `toFile` respectivamente, facilitando el uso de código antiguo.

Usaremos `Paths` para obtener un `Path`:

```
Path ruta1 = Paths.get("/tmp/file.txt"); // UNIX

Path ruta2 = Paths.get("D:/data/file.txt"); // Windows

// Mediante partes de la ruta

Path ruta3 = Paths.get("/tmp", "file.txt");

Path ruta4 = Paths.get("D:", "data", "file.txt");

Path ruta5 = Paths.get("D:/data", "file.txt") ;
```

Realmente es la forma abreviada de:

```
Path ruta1 = FileSystems.getDefault().getPath("/tmp/file.txt");
```

Algunos métodos útiles de `Path` son:

- `equals()`.
- `isAbsolute()`.
- `startsWith()`.
- `endsWith()`.
- `resolve()`: crea una ruta añadiendo la ruta relativa que se recibe como argumento a la que tiene el objeto `Path`.
- `relativize()`: crea una ruta relativa para localizar el elemento que se pase como argumento, desde la ruta que tiene el objeto `Path`.

### 1.4.2. Métodos más importantes de `java.nio.file.Files`.

Los más importantes son:

- Creación:
  - `createFile()`: crea un archivo vacío si no existía previamente.
  - `createDirectory()`: crea un directorio si no existía previamente.
  - `createDirectories()`: crea un directorio si no existía previamente y todos los que se encuentren en su ruta que no existan ya.
- Información:
  - `exists()`: comprueba si un fichero o directorio existe.
  - `notExists()`.
  - `getLastModifiedTime()`: ¿cuándo se modificó por última vez?
  - `getOwner()`: devuelve el propietario.
  - `isDirectory()`.
  - `isExecutable()`.
  - `isHidden()`.
  - `isReadable()`.
  - `isWritable()`.
  - `size()`: tamaño en bytes.
  - `newDirectoryStream()`: devuelve un `DirectoryStream` para poder iterar sobre los elementos del directorio.
- Operaciones sobre el fichero:
  - `setAttribute()`: si queremos modificar los de un archivo. Muy versátil pero complejo tal y como vemos en estos [ejemplos](#). No iremos más allá de saber que existe. Puedes obtener más información [aquí](#).
  - `copy()`: copia todos los bytes de un `Stream` a otro.
  - `move()`: mueve un archivo. Permite varias opciones sobre cómo se comportará la operación.
  - `delete()`: elimina un fichero o directorio si existe y está vacío en el segundo caso.
  - `deleteIfExists()`: comprueba si existe antes de eliminarlo por lo que lanza algunas excepciones menos que el anterior.

**Nota:**

Los *streams*, especialmente los que comunican con elementos de E/S del sistema, debemos cerrarlos. `newDirectoryStream()` devuelve un `DirectoryStream` sobre el que habrá que invocar el método `close()`.

**Para saber más:**

Aunque nosotros no vamos a entrar en detalle, muchas aplicaciones usan ficheros y directorios temporales que el S.O. se encargará de eliminar. Para manejarlos podemos usar los siguientes métodos:

- `createTempFile()`.
- `createTempDirectory()`.

### 1.4.3. Introducción a `java.nio.file.FileStore`.

Se refiere al elemento (dispositivo, partición, volumen, etc.) en el que está almacenado un archivo.

Se obtiene con `Files.getFileStore(Path path)`.

Proporciona información interesante, especialmente:

- `getTotalSpace()`.
- `getUnallocatedSpace()`.
- `getUsableSpace()`.

## Ejercicios 1.2

En los siguientes ejercicios debes repetir los del bloque anterior pero con elementos del paquete `java.nio.files`. Si en alguno de ellos hay algo que no se puede hacer, explica por qué.

1. Ejercicio 1.1.1
2. Ejercicio 1.1.2
3. Ejercicio 1.1.4
4. Ejercicio 1.1.5
5. Ejercicio 1.1.6

6. Ejercicio 1.1.7. En este caso es interesante conocer la [solución de NIO](#) para recorrer el árbol recursivamente, aunque no entraremos en detalles.
7. Ejercicio 1.1.8
8. Crea un objeto Path con `"/alumno/DAM/ejercicios/hola.txt"` y aplica `relative()` pasándole `"/alumno/DAM/apuntes/adios.txt"`. Muestra y explica el resultado.  
Pásale el resultado anterior a el objeto con el método `resolve()`. Muestra y explica el resultado.

---

## 1.5. Detección y tratamiento de excepciones.

Se estudia en primero pero haremos un breve repaso.

**Excepción:** evento que ocurre durante la ejecución de un programa que y interrumpe el flujo normal de ejecución de las sentencias.

Tipos:

- **Excepción comprobada:** derivadas de un evento que se debe haber tenido en cuenta en una aplicación bien programada. Se gestionan dentro de la aplicación usando bloques `try-catch` o propagándolas con `throws`.

Estas excepciones suelen ser el resultado de condiciones que puedes anticipar y manejar adecuadamente, como la inexistencia de un archivo en una operación de lectura.

Ejemplos: `FileNotFoundException`, `IOException`, `SQLException`, etc.

- **Excepción de tiempo de ejecución o no comprobada:** generalmente producidos por un error de programación como que se pase `null` al constructor de `FileReader` o acceder a un índice inválido en un `array`.

Se podría gestionar en la aplicación pero lo deseable sería arreglar el error de programación que provoca esta situación.

Heredan de `RuntimeException`.

Ejemplos: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException` etc.

- **Error:** condiciones excepcionales externos a la aplicación (ocurren en la JVM) y que no se pueden anticipar ni se deberían intentar gestionar (en casi ningún caso).

Ejemplos: error de lectura en un fichero, que hemos abierto correctamente, por un problema en el HDD, quedarse sin memoria RAM, etc.

**Si no la capturamos** en nuestro programa, es capturada por el gestor de excepciones por defecto que devuelve un mensaje y detiene la ejecución del programa.

Para **lanzar** una en nuestro código usamos `throw`.

Si en un método se puede **producir** o **lanzar** una excepción se puede:

- Gestionarla internamente.
- Indicar que se propagará con la palabra reservada `throws`.

Todas las excepciones derivan de `Exception` que a su vez lo hace de `Throwable`.

### 1.5.1. Captura de excepciones.

Tres palabras reservadas:

- `try`.
- `catch`: especifica el o los tipos de excepción que maneja. Puede haber varios; si entra en uno ya no lo hace en los siguientes.
- `finally`: opcional. El código que contiene se ejecuta tanto si sucede la excepción como si no.

Se puede usar `Exception` como tipo genérico pero no es recomendable a no ser que aparezca como última opción en una cadena de `catch` a modo de salvaguarda.

Se pueden anidar bloques `try`:

```
try {
    //Código que puede generar excepciones
    try {
        //más código que puede generar excepciones
    } catch (excepcionInt1 e1){
        //manejo de la excepcion1
    }
} catch (excepcionExt1 e2){
    //manejo de la excepcion1
} catch (excepcionExt2 e3){
    //manejo de la excepcion2
}
// etc .....
finally {
    // se ejecuta después de try o catch
}
```

Podemos capturar varias excepciones en el mismo catch:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
}
```

También es posible relanzar la misma excepción u otra(s):

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

### 1.5.2. Nuestras propias excepciones.

También puedes crear tus propias excepciones para manejar casos específicos en tu aplicación.

Generalmente, heredan de la clase `Exception` para excepciones comprobadas o de `RuntimeException` para excepciones no comprobadas (mucho menos habitual).

Es una buena forma de poder diferenciar situaciones sin necesidad de filtrar usando



condiciones.

```
public class MiExcepcionPersonalizada
    extends Exception {
    // tu código aquí
}
```

#### 1.5.2.1. ¿Es preferible usar muchas excepciones personalizadas o una sola y diferenciar situaciones usando condiciones?

La elección entre usar muchas excepciones personalizadas o una sola con lógica condicional (if-else) para diferenciar situaciones depende de varios factores como la complejidad del problema, la legibilidad del código y la mantenibilidad.

Ventajas de múltiples excepciones personalizadas:

- Legibilidad: El código es más fácil de entender porque cada excepción representa una condición de error específica.
- Mejor manejo de errores: Puedes manejar diferentes tipos de errores de forma más específica usando diferentes bloques catch.
- Extensibilidad: Es más fácil añadir nuevas condiciones de error en el futuro.
- Desacoplamiento: Al separar distintos errores en clases diferentes, reduces el acoplamiento<sup>3</sup> en tu código.

Si usas una única clase de excepción personalizada y utilizas condicionales (if-else) para manejar diferentes tipos de errores, estás acoplando la lógica de manejo de errores en una única clase o bloque catch. Esto significa que cualquier cambio en cómo se manejan los errores tendrá que hacerse en un solo lugar, lo que puede hacer que ese código sea más complejo y difícil de mantener.

Por otro lado, si tienes diferentes clases para diferentes tipos de errores (bajo desacoplamiento), puedes cambiar la lógica de manejo de un tipo específico de error sin afectar a los otros. Cada clase de excepción se encarga de un tipo específico de error, y eso la hace más reutilizable y más fácil de mantener.

---

<sup>3</sup>En el contexto de diseño de software se refiere a la dependencia entre dos o más módulos, clases o componentes. Se considera una mala práctica de diseño.

```
try {  
    // tu código  
} catch (ExcepcionDeArchivoNoEncontrado e) {  
    // manejo específico para archivos no encontrados  
} catch (ExcepcionDePermisos e) {  
    // manejo específico para errores de permisos  
}
```

Ventajas de una única excepción personalizada:

- Simplicidad: Menos clases significan menos complejidad en el código.
- Menor sobrecarga: Reduces la cantidad de clases, lo que podría hacer que el código sea ligeramente más rápido en tiempo de ejecución (este beneficio generalmente es marginal).

```
try {  
    // tu código  
} catch (MiExcepcion e) {  
    if (e.getCodigo().equals(MiExcepcion.ARCH_NO_ENCONTRADO)) {  
        // manejo para archivos no encontrados  
    } else if (e.getCodigo().equals(MiExcepcion.NO_PERMISOS)) {  
        // manejo para errores de permisos  
    }  
}
```

Recomendaciones:

- Si las situaciones de error son muy diferentes entre sí y requieren manejo diferente, es mejor usar excepciones personalizadas para cada una.
- Si las diferentes situaciones de error se manejan de manera bastante similar y solo varían en detalles, una única excepción personalizada podría ser suficiente.
- En general, la legibilidad y la mantenibilidad del código suelen ser más importantes que tener unas pocas clases menos, por lo que suele ser preferible usar múltiples excepciones personalizadas para hacer que el código sea más claro y fácil de mantener.

### 1.5.3. Métodos útiles de Throwable.

- `getMessage`.
- `getLocalizedMessage`.
- `printStackTrace`.
- `toString`.

#### Repaso:

Si necesitas repasar el [manejo](#) de excepciones, este enlace te puede ayudar.

También el [trail](#) de la documentación oficial de Java.

### 1.5.4. Excepciones de ficheros.

La documentación de cada método especifica cuáles puede lanzar.

Conviene conocer las que existen en los paquetes que usamos (sección *Exception Summary*):

- [java.io](#).
- [java.nio.file](#).

### 1.5.5. La sentencia *try-with-resources*.

Para recursos que necesitan ser cerrados.

Asegura que se cerrarán al finalizar el bloque.

Se pueden usar como recursos de este caso los objetos de clases que implementen `java.lang.AutoCloseable`, lo que incluye todos los que implementan `java.io.Closeable`.

Usaremos las clases del ejemplo un poco más adelante. Observa como en el ejemplo se declaran dos recursos a la vez, separados por ;

```
static String leePrimeraLineaDeArchivo(String ruta)
    throws IOException {
    try (FileReader fr = new FileReader(ruta);
        BufferedReader br = new BufferedReader(fr)) {
        return br.readLine();
    }
}
```

Esta solución es mejor que la más antigua en la que se cerraban los recursos en finally, ya que si terminaba abruptamente, podrían quedar recursos sin cerrar:

```
static String leePrimeraLineaDeArchivoConFinally(String ruta)
    throws IOException {
    FileReader fr = new FileReader(ruta);
    BufferedReader br = new BufferedReader(fr);

    try {
        return br.readLine();
    } finally {
        br.close();
        fr.close();
    }
}
```

### Ejercicios 1.3

1. Prueba a dividir dos variables. El divisor debe ser cero. ¿Qué sucede? Solucióvalo.
2. ¿Qué salida producirá el siguiente código?

```
class Main {  
    public static void main(String args[]) {  
        try {  
            throw 10;  
        }  
        catch(int e) {  
            System.out.println("Se produjo una excepción: " + e);  
        }  
    }  
}
```

3. ¿Qué salida producirá el siguiente código?

```
class Prueba extends Exception { }  
  
class Main {  
    public static void main(String args[]) {  
        try {  
            throw new Prueba();  
        }  
        catch(Prueba ex) {  
            System.out.println("Se produjo una excepción Prueba.");  
        }  
        finally {  
            System.out.println("Estoy dentro del bloque finally.");  
        }  
    }  
}
```

4. ¿Qué salida producirá el siguiente código?

```
class Base extends Exception {}
class Derivada extends Base {}

public class Main {
    public static void main(String args[]) {
        // código ...
        try {
            // más código ...
            throw new Derivada();
        }
        catch(Base b) {
            System.out.println("Capturada excepción Base.");
        }
        catch(Derivada d) {
            System.out.println("Capturada excepción Derivada.");
        }
    }
}
```

5. ¿Qué salida producirá el siguiente código?

```
class Prueba {
    public static void main (String[] args) {
        try {
            int a = 0;
            System.out.println ("a = " + a);
            int b = 20 / a;
            System.out.println ("b = " + b);
        }
        catch(ArithmeticException e) {
            System.out.println ("Error al dividir entre cero.");
        }
        finally {
            System.out.println ("Estoy dentro del bloque finally.");
        }
    }
}
```

6. ¿Qué salida producirá el siguiente código?

```
class Prueba {
    int contador = 0;

    void excepcionesAnidadas() throws Exception {
        try {
            contador++;

            try {
                contador++;

                try {
                    contador++;
                    throw new Exception();
                }
                catch(Exception ex) {
                    contador++;
                    throw new Exception();
                }
            }
            catch(Exception ex) {
                contador++;
            }
        }
        catch(Exception ex) {
            contador++;
        }
    }

    void mostrar() {
        System.out.println(contador);
    }

    public static void main(String[] args) throws Exception {
        Prueba objPru = new Prueba();
        objPru.excepcionesAnidadas();
        objPru.mostrar();
    }
}
```

## 1.6. Streams y Buffers.

### 1.6.1. Streams.

Se usan en `java.io`. (Figura 1.1).



Figura 1.1: Streams de E/S.

Dos tipos:

- De *bytes*: E/S de 8 en 8 *bits*. Para trabajar con datos binarios. Todos descienden de `InputStream` y `OutputStream`.
- De caracteres: E/S de carácter en carácter. Por defecto Unicode de 16 *bits*. Todos descienden de `Reader` y `Writer`.

IMPORTANTE: todos los *Streams* deben cerrarse. Generalmente *try-with-resources*.

Los *Streams* son bloqueantes: cuando un hilo lee o escribe en un *Stream*, el hilo queda bloqueado hasta que se reciban o se escriban datos.

**Nota:**

No es tema de este curso, pero al trabajar con caracteres es necesario pensar en si necesitaremos tener en cuenta la [internacionalización](#).

### 1.6.2. Buffers y canales.

No bloqueantes.

Un *buffer* es un contenedor para una cantidad determinada de datos.

Hay que comprobar que el *buffer* tiene todos los datos que necesitamos.

Al añadir datos al *buffer* hay que asegurarse de no sobrescribir datos aún sin procesar.

Propiedades de un *buffer*:



- Capacidad: máximo número de elementos.
- Límite: el primer elemento que no debe ser leído o escrito. Dicho de otra forma, el número de elementos "vivos".
- Posición: el índice del siguiente elemento que será leído o escrito.

Íntimamente relacionados con los canales:

- Canales: portales a través de los cuales se realizan las operaciones de E/S. Bidireccionales.
- *Buffers*: origen o destino de dichas operaciones. Unidireccionales.

Según el tipo de operación:

- Salida: colocamos la información en un *buffer*. El buffer se le pasa al canal.
- Entrada: la información es colocada en el *buffer* desde donde se copia en el canal.

Un único hilo puede gestionar múltiples canales de I/O.

Utilizando selectores, un hilo puede monitorizar múltiples canales de entrada o de salida.

### 1.6.3. **Streams con buffer.**

Los *streams* tradicionales suponen mucho trabajo para el S.O. y el disco porque se el invoca con cada lectura o escritura.

Los *Streams* con *buffer* de `java.io` sí son bloqueantes.

En `java.nio` todas las operaciones utilizan *buffers*.

Los *streams* con *buffer* almacenan información por lo que se invoca al S.O. solo cuando:

- es de lectura (*input*) y el *buffer* está vacío.
- es de escritura (*output*) y el *buffer* está lleno.

Existen envoltorios con *buffer* para *streams* sin él:

- Bytes: `BufferedInputStream` y `BufferedOutputStream`.
- Caracteres: `BufferedReader` y `BufferedWriter`.

## 1.7. Lectura y escritura de archivos de texto.

### 1.7.1. Entrada y salida con *Streams* de caracteres.

Las clases Reader (Figura 1.2). y Writer (Figura 1.3). manejan flujos de caracteres *Unicode*.

Dos clases para convertir flujos de bytes en flujos de caracteres:

- InputStreamReader convierte un InputStream en un Reader.
- OutputStreamWriter convierte un OutputStream en un Writer.

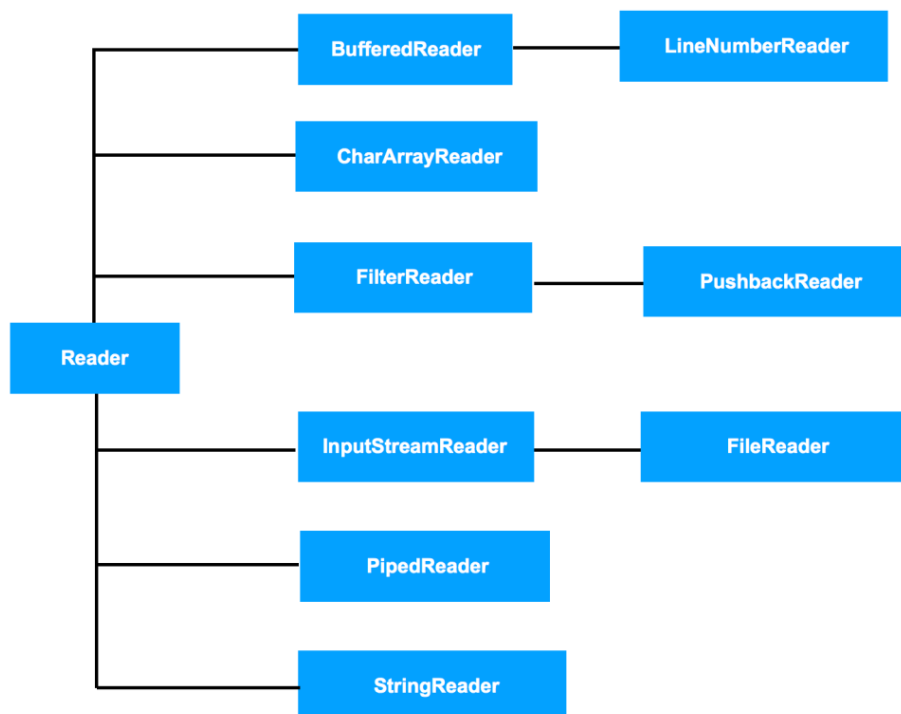
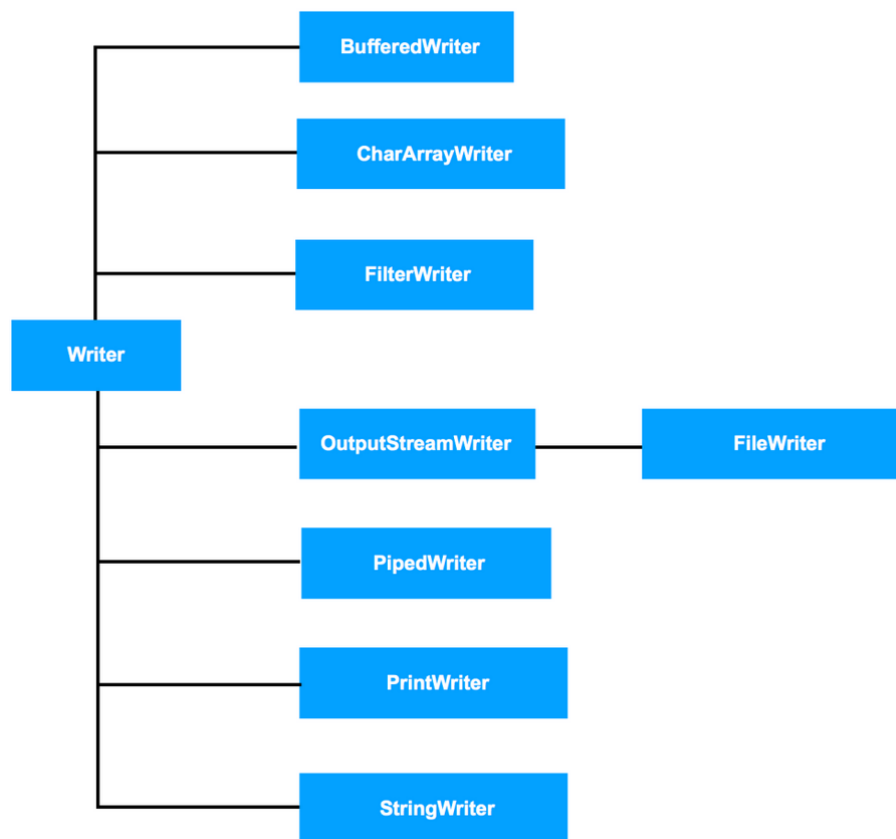


Figura 1.2: Jerarquía Reader. Imagen obtenida [aquí](#).

### 1.7.2. Ejemplos.

El siguiente ejemplo sencillo copia el contenido de un fichero de texto a otro, por lo que sirve de ejemplo de entrada y salida de datos:

Figura 1.3: Jerarquía Writer. Imagen obtenida [aquí](#).

```
public static void main(String[] args) {  
    try (FileReader inputStream =  
        new FileReader("C:\\AccesoADatos\\entrada.txt");  
        FileWriter outputStream =  
            new FileWriter("C:\\AccesoADatos\\salida.txt")) {  
  
        int c;  
        while ((c = inputStream.read()) != -1) {  
            outputStream.write(c);  
        }  
    }  
    catch(FileNotFoundException fnfEx) {  
        System.out.println("El archivo de entrada no existe.");  
    }  
    catch(IOException ioEx) {  
        System.out.println("El archivo de salida no se pudo abrir.");  
    }  
}
```

**IMPORTANTE:**

Suele haber diferentes versiones de los métodos para leer y escribir (como `read` y `write`) por lo que te recomiendo que consultes la documentación de la API de Java con frecuencia para conocer los que más te interesen en cada momento.

Es importante destacar cómo se haría si no usáramos *try-with-resources*:

```
public static void main(String[] args) {
    FileReader inputStream = null;
    FileWriter outputStream = null;

    try {
        inputStream = new FileReader("C:\\AccesoADatos\\entrada.txt");
        outputStream = new FileWriter("C:\\AccesoADatos\\salida.txt");

        int c;
        while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
        }
    }
    catch(FileNotFoundException fnfEx) {
        System.out.println("El archivo de entrada no existe.");
    }
    catch(IOException ioEx) {
        System.out.println("El archivo de salida no se pudo abrir.");
    } finally {
        if (inputStream != null) {
            try {
                inputStream.close();
            } catch (IOException ex) {
                System.out.println("Error cerrando el archivo de entrada.");
            }
        }
        if (outputStream != null) {
            try {
                outputStream.close();
            } catch (IOException ex) {
                System.out.println("Error cerrando el archivo de salida.");
            }
        }
    }
}
```

### 1.7.3. Usando buffers.

Al trabajar con archivos de caracteres generalmente usamos tamaños mayores que un único carácter, por ejemplo líneas.

El siguiente ejemplo es una modificación de la copia de caracteres para que trabaje con líneas. Además usamos un *buffer* en la entrada.

Observa cómo “envolvemos” unos Reader y Writer dentro de otros para obtener la funcionalidad deseada:

- FileReader nos permite leer de archivos, y al meterlo dentro de BufferedReader usamos un *buffer* para hacerlo.
- Usamos FileWriter para escribir en un archivo y al meterlo dentro de PrintWriter obtenemos la capacidad de dar formato (println).

```
public static void main(String[] args) {
    try (BufferedReader inputStream =
        new BufferedReader(
            new FileReader("C:\\AccesoADatos\\entrada.txt"));
        PrintWriter outputStream =
            new PrintWriter(
                new FileWriter("C:\\AccesoADatos\\salida.txt"))) {

        String linea;
        while ((linea= inputStream.readLine()) != null) {
            outputStream.println(linea);
        }
    }
    catch(FileNotFoundException fnfEx) {
        System.out.println("El archivo de entrada no existe.");
    }
    catch(IOException ioEx) {
        System.out.println("El archivo de salida no se pudo abrir.");
    }
}
```

#### IMPORTANTE:

Si en lugar de mostrar lo leído por pantalla, quisiéramos guardarlo en un String, si concatenamos, estaríamos creando un nuevo String todo el rato, ya que son inmutables. Sería más conveniente usar StringBuilder (o StringBuffer que está orientado a trabajar con varios hilos a la vez).

Pero... **¡NO estamos usando buffers para la salida!**, ¿habrá alguna solución? Sí, hacer otro envoltorio:

```
public static void main(String[] args) {
    try (BufferedReader inputStream =
        new BufferedReader(
            new FileReader("C:\\AccesoADatos\\entrada.txt"));
        PrintWriter outputStream =
            new PrintWriter(
                new BufferedWriter (
                    new FileWriter("C:\\AccesoADatos\\salida.txt")))) {

        String linea;
        while ((linea= inputStream.readLine()) != null) {
            outputStream.println(linea);
        }
    }
    catch(FileNotFoundException fnfEx) {
        System.out.println("El archivo de entrada no existe.");
    }
    catch(IOException ioEx) {
        System.out.println("El archivo de salida no se pudo abrir.");
    }
}
```

Se puede definir el tamaño del *buffer* al crearlo, por ejemplo:

- `BufferedWriter(FileWriter, TAM_BUFFER);`
- `BufferedReader(FileReader, TAM_BUFFER);`

#### Repaso:

`PrintWriter` y `PrintStream` (similar para bytes), soportan formatos, al igual que `System.out`. Es útil conocer las diferentes opciones. Si no te acuerdas de primer curso, puedes repasar [aquí](#).

#### 1.7.4. Conjuntos de caracteres.

Aunque no entraremos en detalles es importante tener en cuenta que podríamos estar usando archivos con conjuntos de caracteres diferentes o convirtiendo texto de un conjunto a otro. Para ello se usarían las clases `Charset` y las constantes de `StandardCharsets`.

Por ejemplo para leer de un fichero que tiene un conjunto de caracteres determinado:

```
Charset charset = Charset.forName("UTF_8");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

O para crear y escribir un archivo usando constantes para el conjunto de caracteres:

```
Charset charset = Charset.forName(StandardCharsets.UTF_8);
String s = "Hola a todos";
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

#### 1.7.5. Con la clase `Files`.

Métodos útiles:

- `newBufferedReader()`: crea un buffer para leer del fichero por caracteres usando el charset indicado. Podríamos usar lo visto hasta ahora para *Streams*.
- `newBufferedWriter()`: crea un buffer para escribir en el fichero por caracteres usando el charset indicado. Podríamos usar lo visto hasta ahora para *Streams*.
- `readAllLines()`.
- `write()`: escribe bytes en un fichero.

Podemos leer por líneas:



```
Path ruta = Paths.get("C:/AccesoADatos/entrada.txt");
try {
    // leemos el contenido en Strings.
    List<String> lineas = Files.readAllLines(ruta);

    // Se muestra como una colección (entre []). Podríamos recorrerlo.
    System.out.println("Líneas leídas: \n"+lineas);

    // gestión de excepciones básica para no complicar el ejemplo.
} catch (Exception e) {
    e.printStackTrace();
}
```

**Archivos grandes:**

Para un archivo grande `readAllLines()` no es muy recomendable. Sería mejor obtener un `Reader` con `newBufferedReader()` y trabajar con él.

O escribir un `String` en un archivo:

```
Path path = Paths.get("C:/AccesoADatos/salida.txt");
try {
    String texto = "Hola a todos. \nEsto es un a prueba.";
    byte[] bs = texto.getBytes();
    Path rutaEscrito = Files.write(path, bs);
    System.out.println("Contenido en el fichero:\n"+
        rutaEscrito.getFileName());
} catch (Exception e) {
    e.printStackTrace();
}
```

Existen versiones con conjuntos de caracteres. P.Ej:

```
Files.write(Path,List<String>,Charset);
```

```
Path entrada = Paths.get("entrada.txt");
Path salida = Paths.get("salida.txt");

//Lista de cadenas para leer las lineas
List<String> fileList;
try {
    //Leemos de una vez el archivo de caracteres con java.nio
    fileList = Files.readAllLines(entrada,Charset.forName("UTF-8"));
    //Escribimos una vez el archivo de caracteres con java.nio
    Files.write(salida, fileList,,Charset.forName("UTF-8"));
} catch (IOException e) {
    System.err.format("IOException: %s\n", e);
}
```

### 1.7.6. Opciones de apertura.

Los flujos de `java.nio` pueden configurarse de manera opcional con `OpenOption`.

Existen dos implementaciones en la API oficial:

1. `LinkOption`: para enlaces simbólicos. No entramos en detalles.
2. `StandardOpenOption`: para abrir ficheros con diferentes opciones (entre otras):
  - `READ`: modo lectura.
  - `WRITE`: lectura y escritura.
  - `CREATE`: crea el fichero si no existe. Si ya existe se abre con las otras opciones existentes. Se ignora si va con `CREATE_NEW`.
  - `CREATE_NEW`: crea un archivo si y solo si no existe previamente.
  - `APPEND`: añade al final del fichero; debe usarse con `CREATE` o `WRITE`.
  - `TRUNCATE_EXISTING`: si existe, se abre para escribir y se elimina el contenido previo.
  - `DELETE_ON_CLOSE`: elimina el fichero al cerrarlo.

Las opciones se pueden combinar:

```
String saludo = ";Hola mundo! ";
byte datos[] = saludo.getBytes();
Path ruta = Paths.get("./guardar.txt");

try (OutputStream out = new BufferedOutputStream(
    Files.newOutputStream(ruta, StandardOpenOption.CREATE,
        StandardOpenOption.APPEND))) {
    out.write(datos, 0, datos.length);
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Con la clase Files las opciones se pueden pasar como una lista de argumentos (ejemplo anterior) o un array:

```
Path ruta = Paths.get("./guardar.txt");
Charset charset = Charset.forName("UTF-8");
String s = "Vamos a añadir esta línea al final del archivo";
OpenOption[] options = new OpenOption[2];
options[0] = APPEND;
options[1] = WRITE;
//Creamos un BufferedWriter de java.io utilizando Files de java.nio
try (BufferedWriter writer = Files.newBufferedWriter(ruta, charset,
    options)) {
    writer.newLine();
    writer.write(s, 0, s.length());
} catch (IOException e) {
    System.err.format("IOException: %s\n", e);
}
```

## Ejercicios 1.4

Para probar los ejercicios, hay un archivo *Lorem Ipsum* en el Aula Virtual. Si necesitas usar tildes, etc. puedes incorporarlas a una copia del fichero.

Debes probar cada ejercicio introduciendo el código necesario en el método main y poniendo la comunicación con el usuario que consideres necesaria.

En cada ejercicio debes tener en cuenta la comprobación de los argumentos, el control de posibles errores, etc.

1. Crea una clase `ArchivoTXT` cuyo constructor reciba un `String` y lo guarde como `Path`. Debe comprobar si el `String` es correcto, hace referencia a un archivo (no directorio) y este existe.
2. Añade un método `aVerso` que lea el contenido del archivo y lo devuelva introduciendo un salto de línea después de cada punto.
3. Añade otro método `codifica` que reciba un `String` indicando la ruta de otro fichero (puede existir o no). Debe leer el contenido del archivo original, eliminar todas las vocales y escribir el resultado en el archivo recibido como argumento. Usa únicamente `FileReader` y `FileWriter`.  
Añade otros dos métodos (variaciones de este):
  - Un método `codificaBuffer` usando *buffers*.
  - Un método `codificaFiles` obteniendo las clases con *buffer* a partir de la clase `Files`.
4. Incorpora un método `mover` que reciba otra ruta y mueva el archivo a ella. Si el directorio en el que se encontraba queda vacío, debe eliminarse también.
5. Añade tres métodos:
  - Un método `contar` que cuente el número total de caracteres del archivo.
  - Un método `contarLetras` que cuente el número total de letras del fichero.
  - Un método `contarPuntuación` que cuente el número total de signos de puntuación del fichero.
6. Ahora debes incluir un método `cuentaLineas` que cuente cuántas frases tiene (hasta cada punto), ayudándose con el método `aVerso`.
7. Incorpora otro método `cuentaPalabras` que cuente todas las palabras del fichero.
8. Implementa un método `cuentaVocales` que escriba el número de vocales de cada palabra en un fichero `numVocales.txt` en el mismo directorio en el que se encuentra el fichero original. Cada número debe ir seguido de un espacio. Se deben tener en cuenta tanto mayúsculas como minúsculas pero se contarán juntas. Es decir una *a* y una *A* incrementarán el mismo contador.
9. Modifica el ejercicio anterior para que tenga en cuenta las vocales con tilde y la *u* con diéresis.
10. Implementa el método `frecuenciaLetras` que muestre la frecuencia de las letras (a-z incluidas mayúsculas) del fichero.
11. Crea una clase de algo que te llame la atención. La clase debe incluir al menos un nombre (`String`), otro `String`, un entero, un número de coma flotante, un booleano. Por ejemplo los datos de una persona o de algún personaje de videojuegos, etc.  
Crea varios objetos de esa clase y escríbelos en un archivo.

Los datos se guardarán en un archivo .csv separados por punto y coma.

Prueba a abrir el fichero con una hoja de cálculo y comprobar que se cargue cada elemento en una fila y cada campo en una celda de la misma.

12. Crea un método que permita escribir un objeto de la clase del ejercicio 1.4.11 al final del archivo que ya tienes creado.
13. Lee los objetos del archivo del ejercicio 1.4.12 y muéstralos por pantalla de uno en uno y con un texto delante de cada campo que indique qué es. Puedes sobrescribir el método `toString()` para ayudarte. El archivo se debe abrir en modo lectura.

Prueba a escribir un nuevo elemento, ¿qué sucede?

---

## 1.8. Lectura y escritura de archivos binarios.

Almacenan dígitos binarios que no son legibles directamente por el usuario.

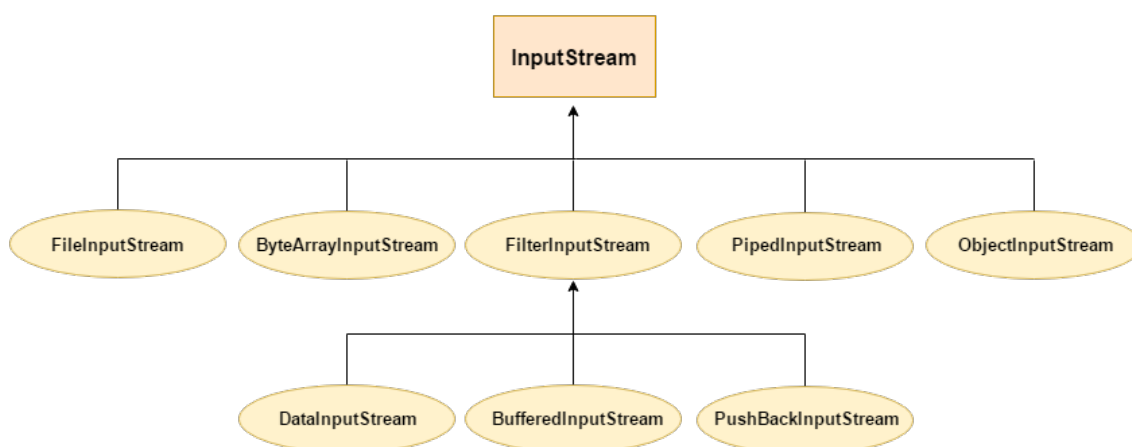
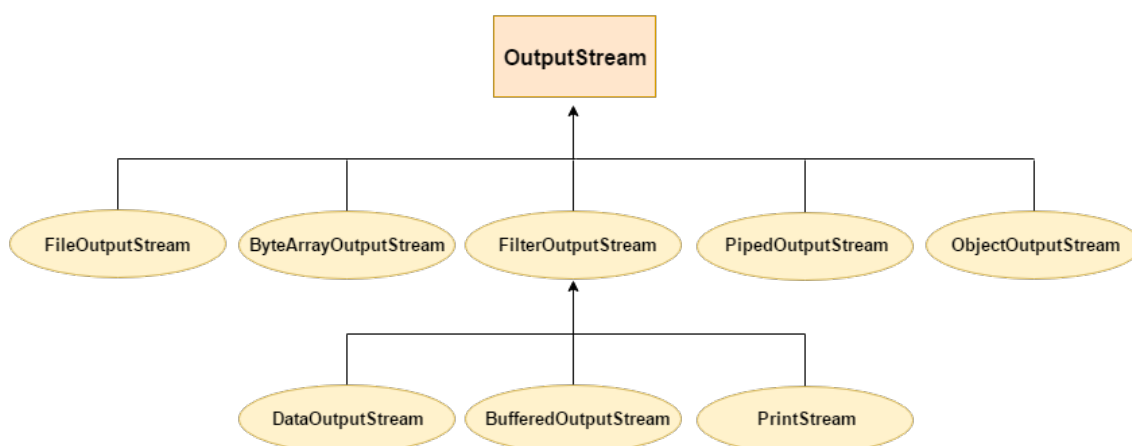
Todos los *streams* de bytes heredan de `InputStream` y `OutputStream`.

### 1.8.1. Entrada y salida con *Streams* de bytes.

Clases básicas:

- `InputStream`: permite entradas de distintas fuentes como un array de bytes, objeto `String`, un fichero, una tubería, una secuencia de otros flujos, una conexión de Internet, etc. (Figura 1.4).
  - `ByteArrayInputStream`: permite usar un espacio de almacenamiento intermedio de memoria
  - `StringBufferInputStream`: convierte un `String` en un `InputStream`.
  - `FileInputStream`: leer información de un fichero.
  - `PipedInputStream`: implementa el concepto de tubería al conectarse a un `PipedOutputStream`. Será el extremo en el que se reciben los datos.
  - `FilterInputStream`: proporciona funcionalidad útil a otras clases `InputStream`. Es como un envoltorio del que heredar para modificar la funcionalidad según nuestras necesidades.
  - `SequenceInputStream`: combina dos o más objetos `InputStream` en un solo.
- `OutputStream`: permite entradas a distintas fuentes como array de bytes, un fichero o una tubería. (Figura 1.5).

- `ByteArrayOutputStream`: crea un espacio de almacenamiento intermedio en memoria donde se enviarán los datos.
- `FileOutputStream`: enviar información a un fichero.
- `PipedOutputStream`: la información que se desee escribir aquí acaba automáticamente como entrada del `PipedInputStream` asociado. Implementa el concepto de tubería al conectarse a un `PipedInputStream`. Será el extremo por el que se envían los datos.
- `FilterOutputStream`: proporciona funcionalidad útil a otras clases `OutputStream`. Es como un envoltorio del que heredar para modificar la funcionalidad según nuestras necesidades.

Figura 1.4: Jerarquía `InputStream`. Imagen obtenida [aquí](#).Figura 1.5: Jerarquía `Output`. Imagen obtenida [aquí](#).

#### Métodos útiles de `InputStream`:

- `read()`: lee el siguiente byte.
- `available()`: devuelve una estimación del número de bytes que se pueden leer todavía del `Stream`.

- `close()`.

Métodos útiles de `OutputStream`:

- `write()`: escribe un byte. Existe una variante que escribe un array de bytes.
- `flush()`: vacía el contenido solicitando que se escriba en el destino. No podemos asegurar que se haga.
- `close()`.

<b>Correspondencia entre las clases de flujos de bytes y caracteres.</b>	
--	--

Clase de flujo de bytes	Clase de flujo de caracteres
<code>InputStream</code>	<code>Reader</code>
<code>OutputStream</code>	<code>Writer</code>
<code>FileInputStream</code>	<code>FileReader</code>
<code>FileOutputStream</code>	<code>FileWriter</code>
<code>StringBufferInputStream</code>	<code>StringReader</code>
	<code>StringWriter</code>
<code>ByteArrayInputStream</code>	<code>CharArrayReader</code>
<code>ByteArrayOutputStream</code>	<code>CharArrayWriter</code>
<code>PipedInputStream</code>	<code>PipedReader</code>
<code>PipedOutputStream</code>	<code>PipedWriter</code>
<code>BufferedInputStream</code>	<code>BufferedReader</code>
<code>BufferedOutputStream</code>	<code>BufferedWriter</code>

### 1.8.2. Ejemplos.

Un ejemplo sencillo y básico:

```
public static void main(String[] args) {
    try (FileInputStream entrada =
        new FileInputStream("C:\\AccesoADatos\\entrada.png");
        FileOutputStream salida =
            new FileOutputStream("C:\\AccesoADatos\\salida.png")){

        int b;

        while ((b = entrada.read()) != -1) {
            salida.write(b);
        }
    }
    catch(FileNotFoundException fnfEx) {
        System.out.println("El archivo de entrada no existe.");
    }
    catch(IOException ioEx) {
        System.out.println("El archivo de salida no se pudo abrir.");
    }
}
```

Se pueden "envolver" *streams* en otros (por ejemplo con *buffer*) igual que en los casos de texto.

### 1.8.3. Con la clase Files.

Métodos útiles:

- `newInputStream()`: abre un stream para leer del archivo. Podríamos usar lo visto hasta ahora para *Streams*.
- `newOutputStream()`: abre un stream para escribir en el archivo. Podríamos usar lo visto hasta ahora para *Streams*.
- `readAllBytes()`.
- `write()`: escribe bytes en un fichero.

Podemos leer por bytes:



```
Path ruta = Paths.get("C:/AccesoADatos/entrada.txt");
try {
    // leemos el contenido en bytes
    byte[] bs = Files.readAllBytes(ruta);

    System.out.println("Bytes leídos: \n"+new String(bs));

    // gestión de excepciones básica para no complicar el ejemplo.
} catch (Exception e) {
    e.printStackTrace();
}
```

#### 1.8.4. **Streams de datos: leer y escribir datos primitivos.**

Dos clases:

- `DataInputStream`.
- `DataOutputStream`.

Métodos útiles de `DataInputStream`:

- `readBoolean()`
- `readByte()`
- `readChar()`
- `readDouble()`
- `readFloat()`
- `readInt()`
- `readLong()`
- `readShort()`
- `readUnsignedByte()`
- `readUnsignedShort()`
- `readUTF()`
- `skipBytes(int n)`

Métodos útiles de `DataOutputStream`:

- `flush()`

- `write(int b)`
- `writeBoolean(boolean v)`
- `writeByte(int v)`
- `writeChar(int v)`
- `writeChars(String s)`
- `writeDouble(double v)`
- `writeFloat(float v)`
- `writeInt(int v)`
- `writeLong(long v)`
- `writeShort(int v)`
- `writeUTF(String str)`

#### 1.8.4.1. Ejemplo: jugadores NFL.

Se usa una clase `JugadorNFL` con tres campos:

- `String nombre`
- `short numero`
- `boolean lesionado`

**Nota:**

Para mantener el ejemplo sencillo, no se comprueban argumentos, se realiza una gestión básica de excepciones, etc.

Y una clase `GestorArchivoJugadores` con métodos estáticos para gestionar la lectura y escritura de objetos `JugadorNFL`.

Método para escribir un jugador:

```
public static void escribirJugador(String archivo, JugadorNFL jugador)
    throws IOException {

    try (DataOutputStream archJugadores =
        new DataOutputStream(new FileOutputStream(archivo))) {

        archJugadores.writeUTF(jugador.getNombre());
        archJugadores.writeShort(jugador.getNumero());
        archJugadores.writeBoolean(jugador.isLesionado());
    }
}
```

Método para escribir una lista de jugadores:

```
public static void escribirJugadores(String archivo,
    List<JugadorNFL> jugadores) throws IOException {

    try (DataOutputStream archJugadores =
        new DataOutputStream(new FileOutputStream(archivo))) {

        for(JugadorNFL jugador: jugadores) {
            archJugadores.writeUTF(jugador.getNombre());
            archJugadores.writeShort(jugador.getNumero());
            archJugadores.writeBoolean(jugador.isLesionado());
        }
    }
}
```

Método para leer una lista de jugadores:

```
public static List<JugadorNFL> leerJugadores(String archivo)
    throws IOException {

    List<JugadorNFL> jugadores = new ArrayList<>();

    try (DataInputStream archJugadores =
        new DataInputStream(new FileInputStream(archivo))) {

        while(true) {
            String nombre = archJugadores.readUTF();
            short num = archJugadores.readShort();
            boolean lesionado = archJugadores.readBoolean();

            JugadorNFL jugador = new JugadorNFL(nombre, num, lesionado);
            jugadores.add(jugador);
        }
    }
    catch(EOFException ex) {
        // alcanzado el final del archivo.
    }

    return jugadores;
}
```

Realmente en el caso anterior solo sería correcto capturar EOFException en la primera de las lecturas, si saltase en el resto, sería un error que indicaría que la estructura del fichero no es correcta:

```
public static List<JugadorNFL> leerJugadores(String archivo)
    throws IOException, FormatoArchivoIncorrectoEx {

    List<JugadorNFL> jugadores = new ArrayList<>();

    try (DataInputStream archJugadores =
        new DataInputStream(new FileInputStream(archivo))) {

        while(true) {
            String nombre = archJugadores.readUTF();

            try {
                short num = archJugadores.readShort();
                boolean lesionado = archJugadores.readBoolean();
            }
            catch(EOFException ex) {
                throw new FormatoArchivoIncorrectoEx();
            }

            JugadorNFL jugador = new JugadorNFL(nombre, num, lesionado);
            jugadores.add(jugador);
        }
    }
    catch(EOFException ex) {
        // alcanzado el final del archivo.
    }

    return jugadores;
}
```

Lo anterior es una situación **normal** ya en la que aprovechamos una excepción para detectar una situación determinada. Aún así, se podría hacer con el método `available()` de `InputStream`:

```
public static List<JugadorNFL> leerJugadores(String archivo)
    throws IOException, EOFException {

    List<JugadorNFL> jugadores = new ArrayList<>();

    try (DataInputStream archJugadores =
        new DataInputStream(new FileInputStream(archivo))) {

        while(archJugadores.available() > 0) {
            String nombre = archJugadores.readUTF();
            short num = archJugadores.readShort();
            boolean lesionado = archJugadores.readBoolean();

            JugadorNFL jugador = new JugadorNFL(nombre, num, lesionado);
            jugadores.add(jugador);
        }
    }

    return jugadores;
}
```

Probamos el método creado:

```
public static void main(String[] args) {
    String archivoJugadoresNFL = "C:/AccesoADatos/jugadoresNFL.dat";
    List<JugadorNFL> jugadores = new ArrayList<>();
    List<JugadorNFL> jugadoresLeidos;

    jugadores.add(new JugadorNFL("Herbert", (short)10, false));
    jugadores.add(new JugadorNFL("Allen", (short)13, true));
    jugadores.add(new JugadorNFL("James Jr.", (short)3, false));

    try {
        GestorArchivoJugadores.escribirJugadores(archivoJugadoresNFL,
            jugadores);

        jugadoresLeidos = GestorArchivoJugadores.leerJugadores(
            archivoJugadoresNFL);

        for(JugadorNFL jugador : jugadoresLeidos) {
            System.out.println(jugador);
        }
    } catch (IOException ex) {
        System.out.println("Error de acceso al fichero de jugadores.");
        ex.printStackTrace();
    }
}
```

### 1.8.5. Streams de objetos: serialización.

Lo anterior es engorroso y no siempre adecuado. Pej: para guardar datos monetarios se recomienda `BigDecimal` pero es una clase, no un dato primitivo.

Solución: objetos que implementan la interfaz `Serializable`.

Permite convertir objetos en una secuencia de bits que puede ser restaurada posteriormente.

Dos clases:

- `ObjectInputStream`.
- `ObjectOutputStream`.

Aunque incluyen métodos para leer o escribir tipos primitivos, los más destacados (respectivamente) son:

- `readObject()`
- `writeObject(Object obj)`

Si un objeto solo contiene datos primitivos, es sencillo, pero ¿qué sucede si contiene objetos de otras clases? Estos necesitan serializarse a su vez y así respectivamente. (Figura 1.6).

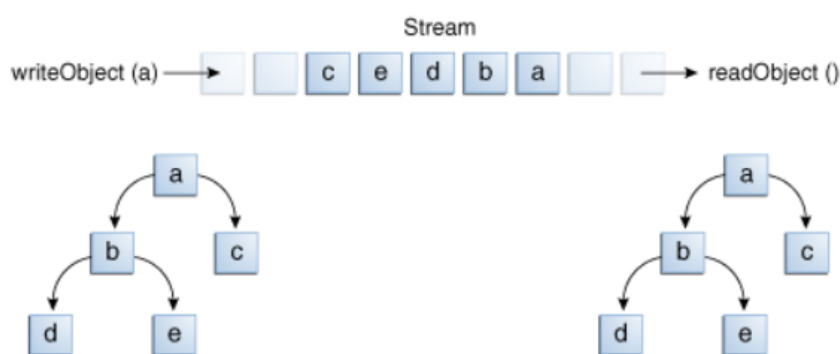


Figura 1.6: Serialización de objetos con referencia a otros objetos. Imagen obtenida [aquí](#).

### IMPORTANTE:

Si un objeto es escrito dos o más veces en un *stream* (flujo), solo se guardará una copia, pero dos (o más referencias) a él. Si leemos las dos (o más) veces que se guardó, obtendremos en las variables referencias al mismo objeto.

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);

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

`ob1` y `ob2` guardarían referencias a un único objeto. Si escribiéramos las copias en flujos diferentes, sí lo estaríamos duplicando.



### 1.8.5.1. Implementar Serializable.

No tiene métodos ni campos que debamos implementar.

Problema: si se cambia la **clase** en el emisor, el receptor puede tener una copia más antigua. → reconstrucción de la clase en el lado que recibe sería imposible → se recomienda añadir un campo:

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

cuyo valor deberíamos ir modificando cuando realizáramos cambios fundamentales en el estructura de la misma clase.

### 1.8.5.2. Ejemplo: jugadores NFL "serializables".

Se introducen los siguientes cambios sobre la clase de jugadores:

- Implementa Serializable.
- Añade un campo `static final long serialVersionUID = 1L;`
- Añade un campo `GregorianCalendar fechaContrato;`

Ahora podemos escribir y leer los jugadores de una vez:

```
public static void escribirJugadores(String archivo,
    List<JugadorNFL> jugadores)
    throws IOException {

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

        for(JugadorNFL jugador: jugadores) {
            archJugadores.writeObject(jugador);
        }
    }
}
```

```
public static List<JugadorNFL> leerJugadores(String archivo)
    throws IOException, ClassNotFoundException {

    List<JugadorNFL> jugadores = new ArrayList<>();

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

        while(true) {
            JugadorNFL jugador = (JugadorNFL)archJugadores.readObject();

            jugadores.add(jugador);
        }
    }
    catch(EOFException ex) {
        // alcanzado el final del archivo.
    }

    return jugadores;
}
```

Se puede producir una nueva excepción si el objeto no se puede convertir a uno de la clase esperada:

```
public static void main(String[] args) {
    String archivoJugadoresNFL = "C:/AccesoADatos/jugadoresNFLSer.dat";
    List<JugadorNFL> jugadores = new ArrayList<>();
    List<JugadorNFL> jugadoresLeidos;

    jugadores.add(new JugadorNFL("Herbert", (short)10, false));
    jugadores.add(new JugadorNFL("Allen", (short)13, true));
    jugadores.add(new JugadorNFL("James Jr.", (short)3, false));

    try {
        GestorArchivoJugadores.escribirJugadores(archivoJugadoresNFL,
            jugadores);

        jugadoresLeidos = GestorArchivoJugadores.leerJugadores(
            archivoJugadoresNFL);

        for(JugadorNFL jugador : jugadoresLeidos) {
            System.out.println(jugador);
        }
    } catch (IOException ex) {
        System.out.println("Error de acceso al fichero de jugadores.");
        ex.printStackTrace();
    } catch (ClassNotFoundException ex) {
        System.out.println("Error leyendo jugadores del fichero.");
        ex.printStackTrace();
    }
}
```

### 1.8.5.3. ¿Qué es un Java Bean?

Una [clase](#) que debe cumplir tres condiciones:

- Tener un constructor sin argumentos.
- Sus atributos de clase deben ser privados y ser accesibles mediante métodos get y set.
- Ser serializable.

Muchas *librerías*<sup>4</sup> los usan. Puedes consultar el [tutorial](#) oficial.

<sup>4</sup>libraries

El principal problema que presentan es el constructor vacío ya que permite instanciar objetos sin ningún dato.

#### 1.8.5.4. Problemas de la serialización.

Presenta varios problemas irresolubles:

- El esquema con el que se guarda el objeto es determinado por Java y no tenemos ningún control sobre él. No podemos adaptarlo a nuestras necesidades.
- Se salta constructores y métodos *setters* al crear el objeto (leyendo datos del archivo) por lo que es fácil que se produzcan errores o incluso ataques. En la propia [documentación](#) oficial nos avisan<sup>5</sup>.
- Conseguir convertir objetos serializados de una versión de la clase a otra es muy complicado.
- Es muy poco eficiente.
- Añade una "cabecera" cada vez que se escriben registros en el archivo (ver siguiente punto).

Posibles alternativas:

- Usar `Externalizable`.
- XML.
- JSON.
- [Protocol Buffers](#).

#### 1.8.5.5. Externalizable.

`Externalizable` nos obliga a implementar los métodos de lectura y escritura del objeto:

- `readExternal(ObjectInput in)`
- `writeExternal(ObjectOutput out)`

Estos métodos suplantán cualquier implementación de `writeObject` y `readObject`.

---

<sup>5</sup>Warning: Deserialization of untrusted data is inherently dangerous and should be avoided. Untrusted data should be carefully validated according to the "Serialization and Deserialization" section of the Secure Coding Guidelines for Java SE. Serialization Filtering describes best practices for defensive use of serial filters.

La serialización usa ambas interfaces: `Serializable` y `Externalizable`. Comprueba si implementa `Externalizable` y solo en caso contrario utiliza `Serializable`.

#### 1.8.5.6. Problema: múltiples cabeceras en el mismo archivo.

Cada vez que añadimos registros *serializables* a un archivo, se añade por defecto una cabecera que indica cómo es cada registro.

Si añadimos registros en varias veces, terminaremos con varias cabeceras. Al leer todos los registros se encontrará con una cabecera en el medio y la intentará leer como un registro, produciendo el error:

```
java.io.StreamCorruptedException: invalid type code: AC
```

La solución más fácil pasa por escribir una sola vez en el archivo, pero ¿y si necesitamos añadir registros? Podríamos leer todos los que haya, añadir el nuestro y volver a leerlos.

```
public static void escribirJugadorSerializable(String archivo,
        JugadorNFLSerializable jugador) throws IOException {
    List<JugadorNFLSerializable> jugadores;

    try {
        jugadores = leerJugadoresSerializable(archivo);
    } catch (ClassNotFoundException ex) {
        /* Como ejemplo de uso de excepciones, interpreto que si el
        * archivo tiene un formato incorrecto, puedo borrar el contenido.
        */
        jugadores = new ArrayList<JugadorNFLSerializable>();
    }

    jugadores.add(jugador);

    escribirJugadoresSerializable(archivo, jugadores);
}
```

#### Nota:

Aquí nos explican una alternativa un poco *chapucera* pero más eficiente. No es posible solucionarlo heredando de `ObjectOutputStream` porque el método que se nombra es invocado por el constructor.

## Ejercicios 1.5

1. En la clase del ejercicio 1.4.11 añade un método dinámico que sirva para escribir el propio objeto en un archivo binario que se reciba como argumento. Debe escribirlo al final.
2. Sobre el ejercicio 1.5.1, añade a la clase dos métodos estáticos: uno para escribir una lista de objetos de ese tipo en un fichero binario y otro para leer todos los del fichero que se indique.  
¿Qué sucede si pruebas a leer un archivo con datos incompatibles?
3. Añade un campo fecha y haz *serializable* la clase del ejercicio 1.4.11. Repite los ejercicios 1.5.1 y 1.5.2 acordemente.
4. Refactoriza el ejercicio 1.5.3 para que implemente *Externalizable* en lugar de *Serializable*.

Ten en cuenta que gran parte de lo que tienes que hacer en los métodos a implementar es lo que hiciste en los ejercicios 1.5.1 y 1.5.2.

¿Qué haces con el campo fecha?

## 1.9. Organización de ficheros y métodos de acceso.

El acceso rápido a la información (consulta) es muy importante pero también lo son otros factores como la facilidad de modificar la información contenida. (Figura 1.7).

### Nota

La siguiente clasificación se aplica a este tipo de estructuras<sup>a</sup> en memoria. Cuando su aplicación en archivos sea diferente, se indicará.

<sup>a</sup>organizaciones planas (no jerárquicas) de registros.

### 1.9.1. Organización secuencial.

Registros almacenados de forma continua. (Figura 1.8).

Para acceder a un registro es necesario recorrer el resto de uno en uno hasta llegar a él.

Contienen una marca de fin de fichero (EOF<sup>6</sup>).

<sup>6</sup>End Of File.

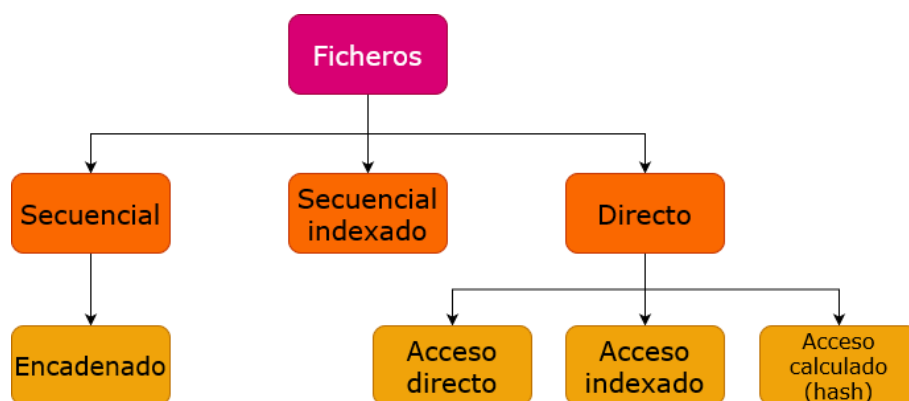


Figura 1.7: Ficheros según el método de acceso.



Figura 1.8: Fichero secuencial. Imagen obtenida del curso MEFP-IFCS03-BD del MEFP.

Generalmente se añaden registros al final del fichero → si se quiere mantener un orden hay que reorganizar el archivo entero (muy costoso).

#### 1.9.1.1. Organización secuencial encadenada.

Cada registro almacena un puntero (dirección de memoria) con la dirección del registro siguiente. Esto elimina la necesidad de añadir registros al final del archivo.

Solo puede usarse en **soportes** que permitan acceso directo.

#### 1.9.2. Organización directa o aleatoria.

Clave: permite localizar de forma rápida el registro.

Hay tres subtipos (Figura 1.8):

- Acceso directo: la clave indica la posición (dirección lógica) del registro.
- Acceso indexado: a cada clave se le asocia una posición en una tabla de claves. La organización de la tabla de claves suele ser secuencial.

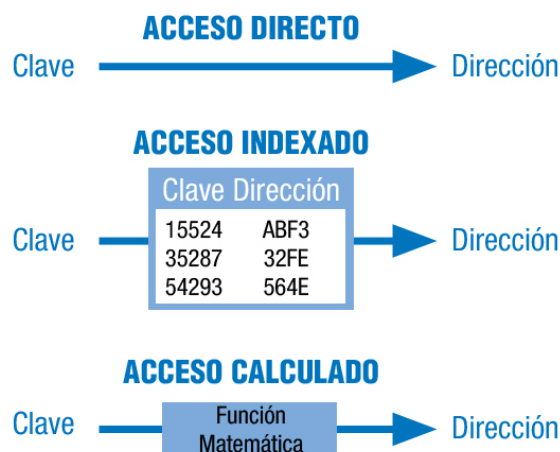


Figura 1.9: Ficheros de acceso directo. Imagen obtenida del curso MEFP-IFCS03-BD del MEFP.

- Acceso calculado (hash): la posición se obtiene al aplicar una serie de cálculos matemáticos a la clave. Puedes obtener una idea más concreta [aquí](#).

Las consultas y borrados son mucho más ágiles que en los secuenciales.

### 1.9.3. Organización secuencial indexada.

A medio camino entre los ficheros secuenciales y los de acceso indexado.

Tiene dos estructuras o zonas:

- Zona de registros: se organiza en segmentos (bloques de registros).
- Zona de índices: una tabla como la del acceso directo pero en la que cada clave indica la posición de un segmento, no de un registro → varios registros están en la misma posición.

La diferencia con el **acceso directo** es que en la **organización secuencial indexada** al obtener la posición, nos lleva a un segmento en el que habrá que buscar el registro de forma secuencial.

### 1.9.4. Operaciones sobre ficheros.

Sobre el fichero:

- Creación del fichero: se le asigna un nombre que servirá para referirse a él en el futuro. Este proceso se realiza una sola vez.
- Apertura del fichero: necesario antes de operar con el contenido.



- Cierre del fichero: al terminar de operar.
- Lectura de datos: transfiere información del fichero a la memoria principal, normalmente a través de alguna variable o variables.
- Escritura de datos: transfiere información de la memoria (por medio de variables) al fichero.

Sobre un registro:

- Alta: añade un nuevo registro.
- Baja: elimina un registro. Antes se necesita localizar el registro. Puede ser:
  - Lógica: cambiando el valor de algún campo del registro.
  - Física: en el mismo fichero o en otro nuevo (copia sin los datos que se desean eliminar, se renombra para sustituir el original).
- Modificación: cambiar parte del contenido de un registro. Antes se necesita localizar el registro.
- Consulta: acceso en el fichero un registro concreto.

### 1.9.5. La clase **RandomAccessFile**.

Permite desplazarse en número de *bytes*.

Puntero que indica la posición actual en el fichero y que inicialmente se coloca al principio, en cero.

Llamadas a los métodos `read()` y `write()` ajustan el puntero según la cantidad de bytes leídos o escritos.

Métodos útiles:

- Métodos de escritura y lectura de tipos primitivos.
- `getFilePointer()`
- `length()`
- `setLength(long newLength)`
- `seek(long pos)posicion`: coloca el puntero en la posición indicada, desde el inicio del archivo.
- `skipBytes(int n)`: desplaza el puntero el número de bytes indicados, desde la posición actual .

**PROBLEMA:** ¿cuántos bytes ocupa un String? ¿y un objeto que contenga Strings, objetos de otras clases, etc.?

Para Strings podemos solucionarlo con `StringBuilder`:

```
...
nombreSB = new StringBuilder(jugador.getNombre());
nombreSB.setLength(TAM_NOMBRE);
salida.writeChars(nombreSB.toString());
...
```

Para leer un objeto en una posición determinada basta con multiplicarla por el número de *bytes* que ocupa cada objeto. P.Ej: para leer el décimo registro multiplicaríamos el tamaño de cada registro por 9.

La clase `Files` proporciona acceso aleatorio a través de la interfaz `SeekableByteChannel` del paquete `java.nio`.

Bastante más compleja para lo que queremos así que no la veremos.

## 1.10. Data access object (DAO).

Interfaz abstracta para persistencia de objetos.

Proporciona operaciones sin exponer los mecanismos propios de cada tipo de almacenamiento.

Proporciona independencia entre la aplicación y el almacenamiento de datos.

### Ejercicios 1.6

1. Piensa y describe brevemente cómo harías cada una de las operaciones sobre registros en las siguientes organizaciones de ficheros:
  - a) Secuencial.
  - b) Secuencial encadenada.
  - c) Directa o aleatoria.
  - d) Secuencial indexada.
2. Completa la clase `Pokemon` para que sea un *Java Bean* que permita almacenar los siguientes atributos:
  - Nombre.
  - Nivel.

- Vida.
- Ataque.
- Defensa.
- Ataque Especial.
- Defensa Especial.
- velocidad.

Si (como yo) no tienes ni idea de Pokemons, aquí tienes una [base de datos](#) que puede ayudarte.

Debes controlar los valores de los parámetros, gestionar posibles errores, etc.

La clase Pokemon debe implementar Externalizable.

3. Crea una clase PokemonDAOFile que implemente la interfaz PokemonDAO que hay en el aula virtual.
4. Crea pruebas para todos los métodos. Añade al menos 10 pokemons al archivo antes de hacer cualquier otra prueba.

---

## 1.11. Bibliografía.

- Carlos Tessier Fernández (2021). Curso [Acceso a Datos](#).
- Curso [MEFP-IFCS02-AD](#). Ministerio de Educación y Formación Profesional.
- Santiago Faci y Fernando Valdeón (2022). Curso [Acceso a datos](#). abrilCode.
- Documentación de la [API](#) de Java.
- [Lesson: Exceptions](#). The Java™ Tutorials.
- [Lesson: Basic I/O](#). The Java™ Tutorials.
- [Java NIO Buffer Tutorial](#). HowToDoInJava.
- [Java I/O Tutorial](#). javaTpoint.
- [Java Files - java.nio.file.Files Class](#). Digital Ocean.
- [Introduction to Java NIO with Examples](#). GeeksforGeeks.
- [Java NIO Tutorials](#). Java Code Geeks.
- [XML Database](#). Wikipedia.
- [Java: what exactly is the difference between NIO and NIO.2?](#). Stack Overflow.
- [Java Object Serialization Specification](#). Oracle.
- [Why Java serialization is flawed](#). StackOverflow.