

AD02.- Manejo de ficheros.

Caso práctico



Ana está empezando a cursar la Formación en Centros de Trabajo (FCT).

Ya ha tenido unas reuniones con **Juan y María**, para saber cómo se trabaja en BK programación. Aunque está haciendo el módulo de FCT en esta empresa, ya sabe que a veces tendrá que salir a otras empresas acompañada de sus tutores para ver los requisitos de los sistemas que la empresa tenga que informatizar y, en ocasiones, **Antonio**, quizás también se apunte para echar una mano.

Ana está nerviosa, también ilusionada, y tiene muchas ganas de conocer de cerca la realidad de lo que ha estudiado en clase.

Ahora verá el uso de los conocimientos adquiridos de diferentes módulos, y buscará respuestas a posibles dudas que se vayan planteando.

En clase les habían explicado la importancia de los ficheros en el acceso a datos. -Es importante repasar los conceptos -piensa Ana.

1.- Introducción.



Si estás estudiando este módulo, es probable que ya hayas estudiado el de programación, por lo que no te serán desconocidos muchos conceptos que se tratan en este tema.

Ya sabes que cuando apagas el ordenador, los datos de la memoria **RAM** se pierden. Un ordenador utiliza ficheros para guardar los datos. Piensa que los datos de las canciones que oyes en mp3, las películas que ves en formato avi, o mp4, etc., están, al fin y al cabo, almacenadas en ficheros, los cuales están grabados en un soporte como son los discos duros, **DVD**, pendrives, etc.

Se llama a los datos que se guardan en ficheros **datos persistentes**, porque persisten más allá de la ejecución de la aplicación que los trata. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos, entre otras cosas, cómo hacer con Java las operaciones de crear, actualizar y procesar ficheros.

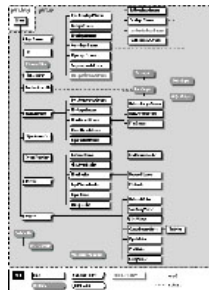
A las operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Las operaciones de E/S en Java las proporciona el paquete estándar de la API de Java denominado `java.io` que incorpora **interfaces**, clases y excepciones para acceder a todo tipo de ficheros.

La **librería** `java.io` contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java. Estas clases de E/S las podemos agrupar fundamentalmente en:

- ✓ Clases para leer entradas desde un flujo de datos.
- ✓ Clases para escribir entradas a un flujo de datos.
- ✓ Clases para operar con ficheros en el sistema de ficheros local.
- ✓ Clases para gestionar la serialización de objetos.

En la imagen puedes ver las clases de las que se dispone en `java.io`.



[.\(link: AD02_CONT R03_clases.png.\)](#)

Autoevaluación

Indica si la afirmación es verdadera o falsa:

Los datos persistentes perduran tras finalizar la ejecución de la aplicación que los trata.

☐ Verdadero ☐ Falso

2.- Clases asociadas a las operaciones de gestión de ficheros y directorios.

Caso práctico



Ana le comenta a **Antonio** -Por lo que nos han comentado, vamos a tener que utilizar bastante los ficheros. ¿Cómo te manejas tú con ellos?

-Pues tan bien como tú -responde **Antonio**, -yo también estudié el módulo de Programación. ¿Es que no recuerdas que en el módulo estudiamos un tema sobre ficheros?

-Sí, pero es que, como había tantos métodos para listar, renombrar archivos, etc., ya casi no me acuerdo; y eso que hace poco que lo estudiamos -contesta **Ana**.

Antonio intenta tranquilizar a **Ana** y le dice que no se preocupe, que en cuanto se les presente la ocasión de tener que programar con ficheros, seguro que no tienen problema y refrescan los conceptos que aprendieron en su día.

En efecto, tal y como dicen Ana y Antonio, hay bastantes métodos involucrados en las clases que en Java nos permiten manipular ficheros y carpetas o directorios.

Vamos a ver la clase `File` que nos permite hacer unas cuantas operaciones con ficheros, también veremos cómo filtrar ficheros, o sea, obtener aquellos con una característica determinada, como puede ser que tengan la **extensión .odt**, o la que nos interese, y por último, en este apartado también veremos como crear y eliminar ficheros y directorios.

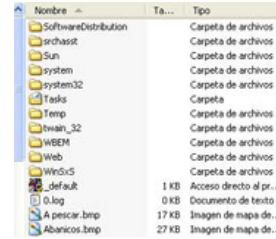


2.1.- Clase File.

¿Para qué sirve esta clase, qué nos permite? La clase `File` proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase `File` representan nombres de archivo, no los archivos en sí mismos.



Nombre	Ta...	Tipo
SoftwareDistribution		Carpeta de archivos
archivos		Carpeta de archivos
Sun		Carpeta de archivos
system		Carpeta de archivos
system32		Carpeta de archivos
Tasks		Carpeta
Temp		Carpeta de archivos
twain_32		Carpeta de archivos
WSEMI		Carpeta de archivos
Web		Carpeta de archivos
WinSxS		Carpeta de archivos
default	1 KB	Acceso directo al pr...
0.jpg	0 KB	Documento de texto
A pensar.bmp	17 KB	Imagen de mapa de...
Abanicos.bmp	27 KB	Imagen de mapa de...

El archivo correspondiente a un nombre puede ser que no exista, por esta razón habrá que controlar las posibles **excepciones**. [\(link: AD02_CONT_R06_carpetas.jpg.\)](#)

Un objeto de clase `File` permite examinar el nombre del archivo, descomponerlo en su rama de directorios o crear el archivo si no existe mediante el método `createNewFile()`;

Para archivos que existen, a través del objeto `File`, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto `File`, podemos hacer las siguientes operaciones con él:

- ✓ **Renombrar** el archivo, con el **método** `renameTo()`. El objeto `File` dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.
- ✓ **Borrar** el archivo, con el método `delete()`. También, con `deleteOnExit()` se borra cuando finaliza la ejecución de la máquina virtual Java.
- ✓ **Crear** un nuevo fichero con un nombre único. El método estático `createTempFile()` crea un fichero temporal y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- ✓ **Establecer** la fecha y la hora de modificación del archivo con `setLastModified()`. Por ejemplo, se podría hacer: `new File("prueba.txt").setLastModified(new Date().getTime());` para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso `prueba.txt`.
- ✓ **Crear** un directorio, mediante el método `mkdir()`. También existe `mkdirs()`, que crea los directorios superiores si no existen.
- ✓ **Listar** el contenido de un directorio. Los métodos `list()` y `listFiles()` listan el contenido de un directorio. `list()` devuelve un vector de `String` con los nombres de los archivos, `listFiles()` devuelve un vector de objetos `File`.
- ✓ **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático `listRoots()`.

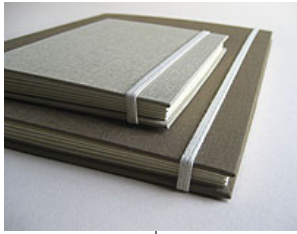
Autoevaluación

Señala si la afirmación es verdadera o falsa:

Podemos establecer la fecha de modificación de un archivo mediante el método `renameTo()`.

☐ Verdadero ☐ Falso

2.1.1.- Ejemplos de la Clase File.



Para crear un objeto File, se puede utilizar cualquiera de los tres constructores siguientes:

```
- File(String directorioyfichero): en Linux: new File("/directorio/fichero.txt");
    en plataformas Microsoft Windows: new
File("C:\\directorio\\fichero.txt");
- File(String directorio, String nombrefichero): new File("directorio",
"fichero.txt");
- File(File directorio, String fichero): new File(new File("directorio"), "fichero.txt");
```

En Linux se utiliza como prefijo de una ruta absoluta "/". En Microsoft Windows, el prefijo de un nombre de ruta consiste en la letra de la unidad seguida de ":" y, posiblemente, seguida por "\\" si la ruta es absoluta. Para utilizar los separadores adecuados independientemente del sistema que estemos utilizando es recomendable utilizar el campo separator.

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método list() que devuelve un array de Strings con los nombres de los ficheros y directorios contenidos en el directorio asociado al objeto File. Para indicar que estamos en el directorio actual creamos un objeto File y le pasamos el parámetro ".":

[VerDir \(link: VerDir.java\)](#)_. (0.01 MB)

El siguiente ejemplo muestra información del fichero o directorio seleccionado. En esta ocasión se pasa la ruta utilizando los argumentos de main().

[VerInf \(link: VerInf.java\)](#)_. (0.01 MB)

Por último, este ejemplo crea un directorio (de nombre NUEVODIR) en el directorio actual, a continuación crea dos ficheros vacíos en dicho directorio y uno de ellos lo renombra. En este caso para crear los ficheros se definen 2 parámetros en el objeto File: File(File directorio, String nombrefich), en el primero indicamos el directorio donde se creará el fichero y en el segundo indicamos el nombre del fichero:

[CrearDir \(link: CrearDir.java\)](#)_. (0.01 MB)

Para borrar un fichero o directorio usamos el método delete(). Para **borrar un directorio** con Java, tendremos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer **recursivamente** el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio e ir borrando con:

```
File[] ficheros = directorio.listFiles();
```

Si el elemento es un directorio, lo sabemos mediante el método isDirectory().

Para saber más

En este enlace puedes ver ejemplos para formatear Strings utilizando printf():

Formatting with printf() (link: <https://www.baeldung.com/java-printstream-printf>)

Autoevaluación

Señala la opción correcta. Con la clase File podemos:

- ☐ (link:)
Crear ficheros temporales.
- ☐ (link:)
Crear directorios.

- ☐ *(link:)*
Renombrar un archivo.
- ☐ *(link:)*
Todas son correctas.

3.- Flujos.

Caso práctico

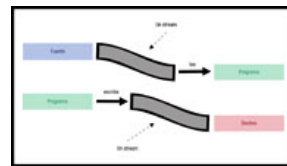


Ana y Antonio saben que van a tener que ayudar a **Juan** y a **María** en labores de programación de ficheros en Java. Así que, además, de la clase `File`, van a necesitar utilizar otros conceptos relacionados con la entrada y salida: los flujos o streams. Ana recuerda que hay dos tipos de flujos: flujos de caracteres y flujos de bytes.

Un programa en Java, que necesita realizar una operación de entrada/salida (en adelante E/S), lo hace a través de un flujo o stream.

Un **flujo** es una **abstracción de todo aquello que produce o consume información**.

La vinculación de este flujo al dispositivo físico la hace el sistema de **entrada y salida** de Java.



[\(link: AD02 CONT R11 flujos.png.\)](#)

Las clases y métodos de E/S que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando. Luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red; liberando al programador de tener que saber con quién está interactuando.

Java define dos tipos de flujos en el paquete `java.io`:

- ✓ **Byte streams** (8 bits): proporciona lo necesario para la gestión de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son `InputStream` y `OutputStream`. Estas dos clases definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan `read()` y `write()` que leen y escriben bytes de datos respectivamente.
- ✓ **Character streams** (16 bits): de manera similar a los flujos de bytes, los flujos de caracteres están determinados por dos clases abstractas, en este caso: `Reader` y `Writer`. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos `read()` y `write()` que leen y escriben caracteres de datos respectivamente.

3.1.- Flujos basados en bytes.

Para el tratamiento de los flujos de bytes, hemos dicho que Java tiene dos clases abstractas que son `InputStream` y `OutputStream`.

Los archivos binarios guardan una representación de los datos en el archivo, es decir, cuando guardamos texto no guardan el texto en sí, sino que guardan su representación en un código llamado .



Las clases principales que heredan de `OutputStream`, para la escritura de ficheros binarios son:

- ✓ `FileOutputStream`: escribe bytes en un fichero. Si el archivo existe, cuando vayamos a escribir sobre él, se borrarán. Por tanto, si queremos añadir los datos al final de éste, habrá que usar el constructor `FileOutputStream(String filePath, boolean append)`, poniendo `append` a `true`.
- ✓ `ObjectOutputStream`: convierte objetos y variables en vectores de bytes que pueden ser escritos en un `OutputStream`.
- ✓ `DataOutputStream`, que da formato a los tipos primitivos y objetos `String`, convirtiéndolos en un flujo de forma que cualquier `DataInputStream`, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "write", como `writeByte()`, `writeln()`, etc.

De `InputStream`, para lectura de ficheros binarios, destacamos:

- ✓ `FileInputStream`: lee bytes de un fichero.
- ✓ `ObjectInputStream`: convierte en objetos y variables los vectores de bytes leídos de un `InputStream`.
- ✓ `DataInputStream`: define diversos métodos para leer datos de tipo primitivo.

En el siguiente ejemplo puedes ver cómo se crea un flujo de salida y otro de entrada a un archivo binario mediante las clases `FileInputStream` y `FileOutputStream`. En primer lugar escribe los bytes en un fichero y después los visualiza. En caso de que queramos añadir bytes al final del fichero usaremos `FileOutputStream` con el segundo parámetro del constructor en `true`.

[EscribirFichBytes \(link: *EscribirFichBytes.java*\).](#)

En los siguientes ejemplos se puede ver cómo se escribe y se lee un archivo binario con `DataOutputStream` y `DataInputStream`

[EscribirFichData \(link: *EscribirFichData.java*\).](#)

[LeerFichData \(link: *LeerFichData.java*\).](#)

Los métodos `read()` o `write()` pueden lanzar la excepción `IOException` por lo que es necesario añadir un `throws IOException` en el `main` o bien utilizar el manejador `try-catch/try with resources`.

Después de realizar las operaciones de lectura y escritura cerraremos el flujo mediante el método `close()`. En caso de que utilicemos `try-with-resources` se cerrará automáticamente.

Para saber más

En el siguiente enlace puedes obtener más información sobre `try with resources`.

[try with resources \(link: *https://www.baeldung.com/java-try-with-resources*\)](https://www.baeldung.com/java-try-with-resources)

Autoevaluación

Señala si la afirmación es verdadera o falsa:

Podemos escribir datos binarios a ficheros utilizando el método `append` de la clase `DataOutputStream`.

☐ Verdadero ☐ Falso

3.1.1.- Objetos en ficheros binarios.

Java nos permite guardar objetos en ficheros binarios; para poder hacerlo, el objeto tiene que implementar la interfaz **Serializable** que dispone de una serie de métodos con los que podremos guardar y leer objetos en ficheros binarios. Los más importantes a utilizar son:

- **Object readObject()** throws **IOException, ClassNotFoundException**: para leer un objeto del **ObjectInputStream**
- **void writeObject(Object obj)** throws **IOException**: para escribir el objeto especificado en el **ObjectOutputStream**

La serialización de objetos de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer y escribir objetos serializables a un stream se utilizan las clases Java **ObjectInputStream** y **ObjectOutputStream** respectivamente. A continuación se muestra la clase **Persona** que implementa la interfaz **Serializable** y, que utilizaremos para escribir y leer objetos en un fichero binario.

Persona ([link: Persona.java](#))

El siguiente ejemplo escribe objetos **Persona** en un fichero. Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación se crea el flujo de salida **ObjectOutputStream**, que es el que procesa los datos y se ha de vincular al fichero de **FileOutputStream**:

EscribirFichObject ([link: EscribirFichObject.java](#))

Para leer objetos **Persona** del fichero necesitamos el flujo de entrada a disco **FileInputStream** y a continuación crear el flujo de entrada **ObjectInputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileInputStream**:

LeerFichObject ([link: LeerFichObject.java](#))

3.1.2.- Problema con ficheros de objetos

Existe un problema con los ficheros de objetos. Al crear un fichero de objetos se crea una cabecera inicial con la información, y a continuación se añaden los objetos. Si el fichero se utiliza de nuevo para añadir más registros, se crea una nueva cabecera y se añaden los objetos a partir de esa segunda cabecera. El problema surge al leer el fichero y en la lectura se encuentra esa segunda cabecera. El programa nos devolverá la excepción `StreamCorruptedException` y no se podrán leer más objetos.

La cabecera se crea cada vez que se pone `new ObjectOutputStream (fichero)`. Para que no se añadan estas cabeceras se puede redefinir la clase `ObjectOutputStream` creando una nueva clase que la herede (`extends`). Y dentro de esa clase se redefine el método `writeStreamHeader()`, que es el que escribe las cabeceras, para que no haga nada. Así, si el fichero ya se ha creado se llamará a ese método de la clase redefinida.

La clase redefinida quedará así:

```
public class MiObjectOutputStream extends ObjectOutputStream
{
    /** Constructor que recibe OutputStream */
    public MiObjectOutputStream(OutputStream out) throws IOException
    {
        super(out);
    }

    /** Constructor sin parámetros */
    protected MiObjectOutputStream() throws IOException, SecurityException
    {
        super();
    }

    /** Redefinición del método de escribir la cabecera para que no haga nada. */
    protected void writeStreamHeader() throws IOException
    {
    }
}
```

Y dentro de nuestro programa, a la hora de abrir el fichero para añadir nuevos objetos se pregunta si ya existe. Si existe, se crea el objeto con la clase redefinida y utilizando el constructor adecuado con parámetro `append` a `true`, y si no existe, el fichero se crea con la clase `ObjectOutputStream`.

3.2.- Flujos basados en caracteres.



Para los flujos de caracteres, Java dispone de dos clases abstractas: `Reader` y `Writer`.

En un programa Java para crear o abrir un fichero se invoca a la clase `File` y a continuación, se crea el flujo de entrada hacia el fichero con la clase `FileReader`. Después se realizan las operaciones de lectura o escritura y cuando terminemos de usarlo lo cerraremos mediante el método `close()`. El siguiente ejemplo lee cada uno de los caracteres del fichero de texto de nombre `LeerFichTexto.java` y los muestra en pantalla, los métodos `read()` pueden lanzar la excepción `IOException`, por ello en `main()` se ha añadido `throws IOException` ya que no se

incluye el manejador `try-catch/try-with-resources`:

[LeerFichTexto \(link: LeerFichTexto.java \)](#)

En el ejemplo anterior, la expresión `((char) i)` convierte el valor entero recuperado por el método `read()` a carácter, es decir, hacemos un cast a `char`. Se llega al final del fichero cuando el método `read()` devuelve `-1`. Para ir leyendo de 20 en 20 caracteres al método `read` le pasaremos un array de caracteres como parámetro.

El siguiente ejemplo escribe caracteres en un fichero de nombre `FichTexto.txt` (si no existe lo crea). Para ello, utilizamos la clase `FileWriter`. Los caracteres se escriben uno a uno y se obtienen de un `String`:

[EscribirFichTexto \(link: EscribirFichTexto.java \)](#)

Como se puede observar en el ejemplo, en vez de escribir los caracteres uno a uno, también podemos escribir todo el array pasando como parámetro e incluso un `String`.

Debes conocer

Para conocer los métodos que proporciona `FileWriter` para escritura puedes consultar directamente la especificación del API de Java 8 (link: <https://docs.oracle.com/javase/8/docs/api/>)

3.2.1.- Búferes en ficheros de texto

Si se usan sólo `FileReader` o `FileWriter`, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro. Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Los `BufferedReader` y `BufferedWriter` añaden un **buffer** intermedio. Cuando se lee o escribe, esta clase controla los accesos a disco. Así, si vamos escribiendo, se guardarán los datos hasta que haya bastantes datos como para hacer una escritura eficiente.

Al leer, la clase leerá más datos de los que se hayan pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez físicamente. Esta forma de trabajar hace los accesos a disco más eficientes y el programa se ejecuta más rápido.

`FileReader` no contiene métodos que nos permitan leer líneas completas, pero `BufferedReader` sí; dispone del método `readLine()` que lee una línea del fichero y la devuelve, o devuelve null si no hay nada que leer o llegamos al final del fichero. También dispone del método `read()` para leer un carácter. Para construir un `BufferedReader` necesitamos la clase `FileReader`:

```
BufferedReader fichero = new BufferedReader(new FileReader(NombreFichero));
```

El siguiente ejemplo lee el fichero `LeerFichTexto.java` línea por línea y las va visualizando en pantalla, en este caso las instrucciones se han agrupado dentro de un bloque try-catch:

[LeerFichTextoBuf \(link: LeerFichTextoBuf.java \)](#)

La clase `BufferedWriter` también deriva de la clase `Writer`. Esta clase añade un buffer para realizar una escritura eficiente de caracteres. Para construir un `BufferedWriter` necesitamos la clase `FileWriter`:

```
BufferedWriter fichero = new BufferedWriter (new FileWriter (NombreFichero));
```

El siguiente ejemplo escribe 10 filas de caracteres en un fichero de texto y después de escribir cada fila salta una línea con el método `newLine()`:

[EscribirFichTextoBuf \(link: EscribirFichTextoBuf.java \)](#)

Por último, la clase `PrintWriter`, que también deriva de `Writer`, posee los métodos `print(String)` y `println(String)` (idénticos a los de `System.out`) para escribir en un fichero. Ambos reciben un `String` y lo imprimen en un fichero, el segundo método salta de línea. Para construir un `PrintWriter` necesitamos la clase `FileWriter`:

```
PrintWriter fichero = new PrintWriter (new FileWriter(NombreFichero));
```

El ejemplo anterior usando la clase `PrintWriter` y el método `println()` quedaría:

[EscribirFichTextoPrint \(link: EscribirFichTextoPrint.java \)](#)

En este último ejemplo utilizamos try-with-resources por lo que no es necesario cerrarlo manualmente.

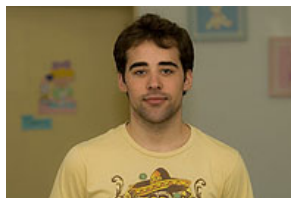
Autoevaluación

Mediante las clases que proporcionan buffers se pretende que se hagan lecturas y escrituras físicas a disco, lo antes posible y cuantas más mejor.

☐ Verdadero ☐ Falso

4.- Formas de acceso a un fichero.

Caso práctico



El momento de programar ha llegado, y **Antonio** se pregunta qué opción será mejor para el acceso a los ficheros: si **acceso secuencial o aleatorio**. ¿Qué uso se le va a dar a estos ficheros?, ¿para qué van a servir cuando la aplicación informática esté en funcionamiento? Esa es la cuestión clave, piensa **Antonio**.

Hemos visto que en Java puedes utilizar **dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio)**. Si bien, y según la literatura que consultemos, a veces se distingue una tercera forma de acceso denominada concatenación, tuberías o pipes.

- ✓ **Acceso aleatorio:** los archivos de acceso aleatorio, al igual que lo que sucede usualmente con la memoria (RAM=Random Access Memory), permiten acceder a los datos en forma no secuencial, desordenada. Esto implica que el archivo debe estar disponible en su totalidad al momento de ser accedido, algo que no siempre es posible.
- ✓ **Acceso secuencial:** En este caso los datos se leen de manera secuencial, desde el comienzo del archivo hasta el final (el cual muchas veces no se conoce a priori). Este es el caso de la lectura del teclado o la escritura en una consola de texto, no se sabe cuándo el operador terminará de escribir.



Citas para pensar

El futuro tiene muchos nombres. Para los débiles es lo inalcanzable. Para los temerosos, lo desconocido. Para los valientes es la oportunidad.

Víctor Hugo.

4.1.- Operaciones básicas sobre ficheros de acceso secuencial.

Como **operaciones más comunes en ficheros de acceso secuencial**, tenemos el acceso para:

- ✓ Crear un fichero o abrirlo para grabar datos.
- ✓ Leer datos del fichero.
- ✓ Borrar información de un fichero.
- ✓ Copiar datos de un fichero a otro.
- ✓ Búsqueda de información en un fichero.
- ✓ Cerrar un fichero.



4.2.- Operaciones básicas sobre ficheros de acceso aleatorio.

A menudo, no necesitas leer un fichero de principio a fin, sino simplemente acceder al fichero como si fuera una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase `RandomAccessFile` para este tipo de entrada/salida.



Esta clase:

- ✓ Permite leer y escribir sobre el fichero, no son necesarias dos clases diferentes.
- ✓ Necesita que le especifiquemos el modo de acceso al construir un objeto de esta clase: sólo lectura o bien lectura y escritura.
- ✓ Posee métodos específicos de desplazamiento como `seek(long posicion)` o `skipBytes(int desplazamiento)` para poder movernos de un registro a otro del fichero, o posicionarnos directamente en una posición concreta del fichero.

Por esas características que presenta la clase, un archivo de acceso directo tiene sus registros de un tamaño fijo o predeterminado de antemano.

La clase posee dos constructores:

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

En el primer caso se pasa un objeto `File` como primer parámetro, mientras que en el segundo caso es un `String`. El modo es: "r" si se abre en modo lectura o "rw" si se abre en modo lectura y escritura.

El ejemplo que se muestra a continuación inserta datos de empleados en un fichero aleatorio. Por cada empleado también se insertará un identificador que coincidirá con el índice +1 con el que se recorren los arrays. La longitud del registro de cada empleado es la misma (36 bytes) y los tipos que se insertan y su tamaño en bytes es el siguiente:

- ✓ Se inserta en primer lugar un entero, que es el identificador, ocupa 4 bytes.
- ✓ A continuación una cadena de 10 caracteres, es el apellido, cada carácter Unicode ocupa 2 bytes, por tanto el apellido ocupa 20 bytes.
- ✓ Un tipo entero que es el departamento, ocupa 4 bytes.
- ✓ Un tipo `Double` que es el salario, ocupa 8 bytes.

Tamaño de otros tipos: `short` (2 bytes), `byte` (1 byte), `long` (8 bytes), `boolean` (1 bit), `float` (4 bytes), etc.

El fichero se abre en modo "rw" para lectura y escritura:

[EscribirFichAleatorio \(link: *EscribirFichAleatorio.java*\).](#)

El siguiente ejemplo toma el fichero anterior y visualiza todos los registros. El posicionamiento para empezar a recorrer los registros empieza en 0, para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento:

[LeerFichAleatorio \(link: *LeerFichAleatorio.java*\).](#)

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero, conociendo su identificador podemos acceder a la posición que ocupa dentro del mismo y obtener sus datos

Autoevaluación

Indica si la afirmación es verdadera o falsa:

Un objeto de la clase `RandomAccessFile` necesita el modo de acceso al crear el objeto.

☐ Verdadero ☐ Falso

Anexo I.- Código de crear un fichero.

([link:](#))

```
try {  
    // Creamos el objeto que encapsula el fichero  
    File fichero = new File("c:\\prueba\\miFichero.txt");  
    // A partir del objeto File creamos el fichero físicamente  
    if (fichero.createNewFile())  
        System.out.println("El fichero se ha creado correctamente");  
    else  
        System.out.println("No ha podido ser creado el fichero");  
} catch (Exception ioe) {  
    ioe.getMessage();  
}
```

Anexo II.- Código de crear un directorio.

[\(link.\)](#)

```
try {  
    // Declaración de variables  
    String directorio = "C:\\prueba";  
    String varios = "carpeta1/carpeta2/carpeta3";  
  
    // Crear un directorio  
    boolean exito = (new File(directorio)).mkdir();  
    if (exito)  
        System.out.println("Directorio: " + directorio + " creado");  
    // Crear varios directorios  
    exito = (new File(varios)).mkdirs();  
    if (exito)  
        System.out.println("Directorios: " + varios + " creados");  
}catch (Exception e){  
    System.err.println("Error: " + e.getMessage());  
}
```

