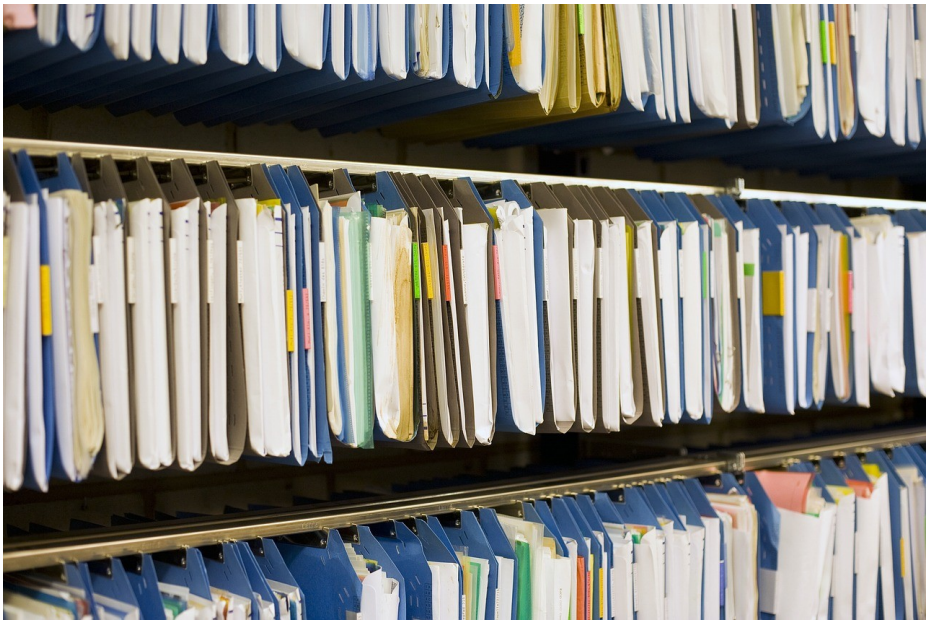
TEMA 6:
FLUJOS Y **FICHEROS**



Ver. Doc. 9.2

CONTENIDO

1 Archivos (ficheros)

Hasta este momento, todas las operaciones de entrada y salida de datos de nuestros programas se han hecho a través del teclado (entrada) y la pantalla (salida). Estos son los dispositivos de entrada y salida por defecto, pero también se pueden **enviar datos hacia un archivo, o recibirlos de él**.

Además, todos los datos que hemos manejado, ya sea mediante tipos de datos simples o estructuras complejas, han estado alojados en la **memoria principal** del ordenador, de manera que al apagar éste, o antes, al terminar el programa, toda esa información se perdía. Como es natural, también es posible almacenar datos en **memoria secundaria**, es decir, en dispositivos tales como discos duros, discos flexibles, discos ópticos, memorias USB, etc. Estas memorias se caracterizan por ser **más lentas** que la memoria principal del ordenador, pero también disponen de **más espacio** de almacenamiento, y **no son volátiles**, es decir, no pierden su contenido al desconectar el ordenador.

Para **almacenar datos en estas memorias secundarias** es necesario **agruparlos** en estructuras que denominaremos **archivos** o **ficheros** (en inglés, *files*)

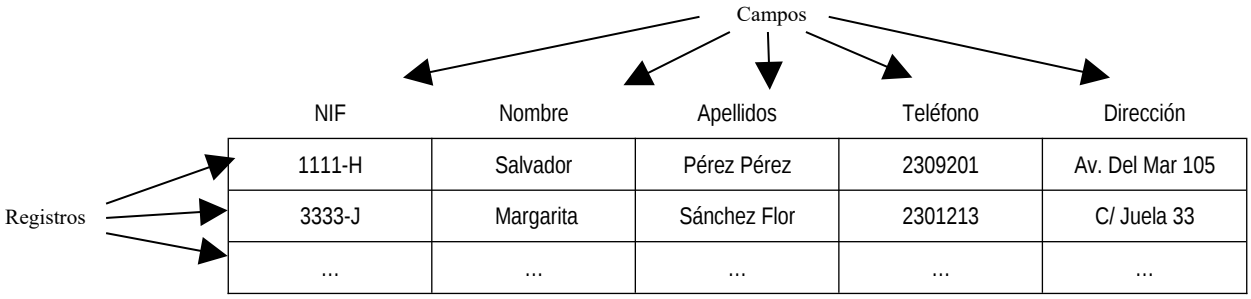
1.1 Conceptos fundamentales: archivos, registros y campos

Un **archivo** o fichero es un conjunto de información relacionada entre sí y estructurada en unidades más pequeñas, llamadas registros.

Cada **registro** debe contener datos pertenecientes a una misma cosa. Además, cada registro es un estructura de datos, es decir, está compuesto de otros datos más simples, que llamaremos campos.

Un **campo** es cada uno de los elementos que constituyen un registro. Cada campo se caracteriza por un identificador que lo distingue de los otros campos del registro, y por el tipo de dato que tiene asociado, que, a su vez, puede ser simple (número entero, carácter, lógico, etc) o compuesto (cadena de caracteres, fecha, vector, etc).

Observa el siguiente ejemplo de fichero. Contiene información relacionada entre sí: los datos personales de un conjunto de personas. Toda esa información está distribuida en registros, que son cada una de las filas de la tabla. Cada registro, por tanto, contiene los datos pertenecientes a una sola persona. Los registros se dividen en campos, que son cada una de las unidades de información que contiene cada registro.



Si el tipo de dato de un campo es complejo, el campo puede dividirse en **subcampos**. Por ejemplo, si un campo contiene una fecha, se puede dividir en tres subcampos que contengan, respectivamente, el día, el mes y el año.

Para diferenciar a un registro de otro es conveniente que alguno de los campos tenga un valor distinto en todos los registros del archivo. Este campo, que identifica unívocamente cada registro, se denomina campo clave o, simplemente, **clave**. En el ejemplo anterior, el campo clave puede ser NIF, ya que será diferente para cada una de las personas que forman el archivo.

1.2 Registros físicos

Un registro físico, también llamado **bloque**, es diferente de los registros que acabamos de ver (que, para diferenciarlos, a veces se denominan registros lógicos)

El **registro físico** es la cantidad de información que el sistema operativo puede enviar o recibir del soporte de memoria secundaria en una operación de escritura o lectura. Esta cantidad depende del hardware.

El registro físico puede ser **mayor** que el registro lógico, con lo cual, en una sola operación de lectura o escritura, se podrían transferir varios registros lógicos. También puede ocurrir lo contrario, es decir, que el registro físico sea de **menor** tamaño que el lógico, lo que haría que para transferir un registro lógico fueran necesarias varias operaciones de lectura o escritura.

Se llama **factor de bloqueo** al número de registros lógicos contenidos en un registro físico.

Como ejemplo vamos a calcular el factor de bloqueo del archivo del epígrafe anterior. Supongamos que el tamaño del registro físico es de 512 bytes (es decir, en una sola lectura o escritura del dispositivo de almacenamiento se pueden transferir 512 bytes) y el registro lógico ocupa 128 bytes, calculados de esta manera[1].

Campo NIF (10 caracteres)	=	10 bytes
Campo Nombre (30 caracteres)	=	30 bytes
Campo Apellidos (40 caracteres)	=	40 bytes
Campo Teléfono (entero largo)	=	8 bytes
Campo Dirección (40 caracteres)	=	40 bytes
TOTAL		*128 bytes*

En estas condiciones, **el factor de bloqueo es 4**, que es el resultado de dividir 512 (tamaño del registro físico) entre 128 (tamaño del registro lógico). En cada registro físico caben exactamente 4 registros lógicos, sin que sobre ningún byte, porque la división de 512 entre 128 es exacta, pero puede ocurrir que no sea así.

Por ejemplo, si el registro lógico ocupase 126 bytes en lugar de 128, en cada registro físico cabrían 4 registros lógicos pero sobrarían 8 bytes. Esto tiene una gran importancia desde el punto de vista del **rendimiento**, ya que cada acceso a la memoria secundaria requiere bastante tiempo y, por tanto, éstos deben reducirse al máximo.

1.3 Tipos de registros

(Nota: cuando hablemos de "registro" a secas nos estaremos refiriendo al registro lógico, no al físico)

Dependiendo de la **longitud de los campos** que forman cada registro podemos **clasificar** éstos en:

1.3.1 Registros de longitud fija

Son los que ocupan siempre el mismo espacio a lo largo de todo el archivo (en el ejemplo anterior, 128 bytes). Dentro de estos registros, podemos encontrar varias posibilidades:

- *Igual número de campos por registro e igual longitud de todos los campos*
- *Igual número de campos por registro y distinta longitud de cada campo, aunque igual en todos los registros*
- *Igual número de campos por registro y distinta longitud de cada campo, pudiendo ser diferente en cada registro*
- *Distinto número de campos por registro y distinta longitud de cada campo en cada registro*

1.3.2 Registros de longitud variable

Aunque es menos habitual, pudiera ocurrir que cada registro del archivo tuviera una longitud propia y diferente del resto. En este tipo de archivos es necesario programar algún mecanismo para averiguar cuál es el principio y el final de cada registro.

[1] Recuerda que el tamaño exacto que requiere cada tipo simple depende del sistema operativo, del compilador y del hardware de la máquina

1.4 Operaciones con archivos

En un archivo se puede realizar operaciones **sobre cada registro individual** o bien **sobre todo el archivo**, es decir, sobre todos los registros a la vez.

1.4.1 Operaciones con registros individuales

- ***Inserción*** *(alta)* → *consiste en añadir un registro al fichero. El registro puede añadirse al final del fichero o entre dos registros que ya existieran previamente.*
- ***Borrado*** *(baja)* → *consiste en eliminar un registro existente.*
- ***Modificación*** → *consiste en cambiar el dato almacenado en uno o varios de los campos del registro*
- ***Consulta*** → *consiste en leer el dato almacenado en uno o varios de los campos del registro.*

1.4.2 Operaciones sobre el archivo completo

Además de manipular cada componente del archivo (registros y campos), también se pueden llevar a cabo **operaciones con la totalidad del archivo**, como:

- ***Creación*** → *La creación del archivo consiste en crear una entrada en el soporte de memoria secundaria y asignarle un nombre para identificar en el futuro a los datos que contiene.*
- ***Apertura*** → *Antes de trabajar con un archivo es necesario abrirlo, creándose así un canal de comunicación entre el programa y el archivo a través del cuál se pueden leer y escribir datos. Los archivos sólo deben permanecer abiertos el tiempo estrictamente necesario.*
- ***Cierre*** → *Es importante cerrar el canal de comunicación con el archivo cuando no va a usarse en un futuro inmediato, porque todos los sistemas limitan el número máximo de archivos que pueden estar abiertos simultáneamente. También es importante porque evita un acceso accidental al archivo que pueda deteriorar la información almacenada en él.*
- ***Ordenación*** → *Permite establecer un orden entre los registros del archivo.*
- ***Copiado*** → *Crea un nuevo archivo con la misma estructura y contenido que el fichero original.*
- ***Concatenación*** → *Consiste en crear un archivo nuevo que contenga los registros de otros dos archivos previamente existentes, de manera que primero aparezcan todos los registros de un archivo y, a continuación, todos los del otro.*
- ***Mezcla*** → *Parecida a la concatenación, pero el archivo resultante contendrá todos los registros de los dos archivos originales mezclados y ordenados.*
- ***Compactación*** → *Esta operación sólo se realiza sobre archivos en los cuales el borrado de registros se ha realizado sin eliminar físicamente el registro, sino únicamente marcándolo como borrado para no procesarlo. Después de la compactación, todos los registros marcados como borrados quedan borrados físicamente, con lo que se libera espacio en el dispositivo de almacenamiento.*
- ***Borrado*** → *Es la operación contraria a la creación, ya que elimina la entrada en el dispositivo de almacenamiento, con lo que se pierde toda la información almacenada en el archivo.*

2 Organización de archivos

La **organización de los archivos** es la forma en que los datos son estructurados y almacenados en el dispositivo de almacenamiento. El tipo de organización se establece durante la fase de creación del archivo y

es invariable durante toda su vida. La organización puede ser secuencial o relativa (o una combinación de ambas), como enseguida veremos.

El **tipo de acceso** al archivo es el procedimiento que se sigue para situarnos sobre un registro concreto para hacer alguna operación con él. Esto es lo que realmente le interesa al programador: cómo acceder a los registros de archivo. El tipo de acceso está condicionado por el tipo de organización física del archivo.

A lo largo de todo **este apartado 2 estudiaremos los tipos de organización**. En los apartados siguientes veremos como se las apaña Java para acceder y manipular los ficheros, y cómo podemos operar desde Java con los distintos tipos de organización.

2.1 Archivos de organización secuencial

La forma más simple de estructura de archivo es el **archivo secuencial**. En este tipo de archivo, los **registros** se sitúan físicamente en el dispositivo en el orden en el que se van escribiendo, **uno tras otro** y sin dejar huecos entre sí. El acceso a los registros también debe hacerse en orden, de modo que para acceder al registro N es necesario pasar primero por el registro 1, luego por el 2, luego por el 3, y así hasta llegar al registo N.

Los archivos secuenciales se utilizaban mucho cuando el soporte de almacenamiento masivo más usual era la cinta magnética. Hoy día siguen siendo utilizados por su **simplicidad** y porque son suficientemente útiles en muchas ocasiones (por ejemplo, en aplicaciones de proceso de lotes). Pero si el programa necesita acceder a registros individuales y no consecutivos, los archivos secuenciales ofrecen un **rendimiento pobre** y son preferibles los archivos directos, que luego veremos.

Los archivos secuenciales tienen un **indicador de posición** (o **cursor**) que señala qué registro fue el último que se accedió. Al abrir el archivo, el indicador se sitúa en el primer campo del primer registro. Cada acceso sobre el archivo **desplazará el indicador de posición** hacia el siguiente registro, hasta que ya no haya más registros que leer.

Cuando un archivo secuencial se abre para **escribir datos** en él, el indicador de posición se sitúa justo **después del último byte** del mismo, de manera que los datos sólo se pueden añadir al final.

La organización secuencial cuenta con varias **ventajas**:

1) Es la más sencilla de manejar para el programador.
2) Si hay que acceder a un conjunto de registros consecutivos, o a todo el archivo, es el método más rápido.
3) No deja espacios entre registro y registro, por lo que se optimiza el uso del espacio en la memoria secundaria.

Pero también tiene algunos **inconvenientes** serios:

- Para consultar datos individuales, hay que recorrer todo el archivo desde el principio. Es decir, el acceso a registros individuales es, en general, lento.
- Las operaciones de inserción y eliminación de registros solo pueden hacerse al final del archivo. Hacerlas con registros intermedios representa mover grandes bloques de información y, por lo tanto, consumir mucho tiempo.

2.2 Archivos de organización relativa: hashing

La organización relativa es más compleja que la secuencial.

Consiste en **guardar físicamente los registros en lugares de la memoria secundaria no consecutivos**. Pero, entonces, ¿cómo podemos encontrar dónde está cada registro?

La única solución es utilizar un **campo clave** de entre todos los del registro. Ese campo clave, que suele ser numérico, permite averiguar la dirección física donde está almacenado el registro en la memoria secundaria mediante un algoritmo de transformación. Por eso, la clave suele denominarse **dirección de memoria lógica**, para distinguirlo de la dirección de memoria física donde efectivamente se encuentra guardado el registro.

Esta transformación de claves para obtener direcciones físicas se denomina ***hashing***. Más abajo encontrarás un ejemplo muy sencillo de hashing que te ayudará a entender todo esto.

Los archivos relativos son **más versátiles** que los secuenciales porque permiten acceder a cualquier parte del fichero en cualquier momento, como si fueran arrays. Las operaciones de lectura y escritura pueden hacerse **en cualquier punto del archivo**.

Los archivos con organización relativa tienen dos variantes: los archivos **directos** y los archivos **aleatorios** (o indirectos). En los siguientes epígrafes estudiaremos cada tipo por separado.

Ejemplo de hashing

Antes de continuar, vamos a tratar de **entender bien la técnica de hashing** con un sencillo **ejemplo**.

Supongamos que un archivo almacenado en una memoria secundaria contiene 5 registros, que llamaremos R1, R2, R3, R4 y R5. En un **archivo secuencial**, los cinco registros estarán almacenados en posiciones consecutivas de la memoria. Si R1 se guarda, por ejemplo, en la dirección 1000 de la memoria secundaria y cada registro lógico ocupa exactamente un registro físico, tendremos que los registros estarán guardados en estas direcciones:

Dirección	1000	1001	1002	1003	1004
Registro almacenado en esa posición	R1	R2	R3	R4	R5

En cambio, si el archivo es **relativo**, cada registro estará almacenado en posiciones no consecutivas de la memoria secundaria. Por ejemplo, podrían estar en estas direcciones:

Dirección	1000	…	1200	…	5720	…	6304	…	6318
Registro almacenado en esa posición	R1	…	R2	…	R3	…	R4	…	R5

El problema con este sistema de almacenamiento es **cómo localizar los registros en la memoria secundaria**. Para eso se utiliza el **hashing**. Cada registro debe tener un campo clave (que denominaremos R1.clave, R2.clave, etc). El hashing consiste en aplicar una función de transformación a cada clave.

Supongamos que las **claves** de los registros de este ejemplo son:

```
R1.clave = 500
R2.clave = 600
R3.clave = 2860
R4.clave = 3152
R5.clave = 3159
```

Entonces, la **función hash** aplicada a este archivo para averiguar la dirección de cada registro ha sido

```
f(clave) = clave x 2
```

Probemos a **aplicar la función hash al primer registro** (R1):

```
f(R1.clave) = 500 x 2 = 1000
```

Efectivamente, aplicando la función hash a la clave de R1 (500), hemos obtenido su dirección de almacenamiento en memoria secundaria (1000).

Si probamos con **otros registros**, esta función hash también nos devuelve la dirección. Por ejemplo, con R3:

```
f(R3.clave) = 2860 x 2 = 5720
```

Si lo compruebas, 5720 es la dirección donde está guardado el registro R3.

2.3 Archivos de organización relativa directa

Entre los archivos con organización relativa los más sencillos son los **directos**.

En ellos, el **campo clave** de cada registro debe ser de tipo **numérico**, e **identifica directamente el registro físico** donde está almacenado. La función hash, en este caso, es la más simple posible, ya que no transforma la clave:

```
f(clave) = clave
```

En el ejemplo anterior, el registro R1 se almacenaría en la dirección 500, el R2 en la 600, el R3 en la 2860, etc, ya que:

```
f(R1.clave) = clave = 500
f(R2.clave) = clave = 600
f(R3.clave) = clave = 2860
```

El valor de la clave está en relación con la **capacidad máxima del dispositivo** de almacenamiento, no pudiendo almacenar registros cuya clave esté por encima de este límite.

En estos archivos no puede haber dos registros con la misma clave, porque ambos ocuparían la misma posición física, solapándose. Esto es lo que se llama una **colisión** y debe ser evitada.

Las **ventajas** de los archivos directos son:

- Permite acceder al archivo de dos maneras: directamente (a través de la clave de cada registro) y secuencialmente.
- Permite realizar operaciones de lectura y escritura simultáneamente.
- Son muy rápidos al tratar registros individuales.

Los **inconvenientes** principales son:

- El acceso secuencial, del principio al fin del fichero, puede ser muy lento porque podemos encontrarnos con muchos huecos, es decir, posiciones que no están siendo usadas. Existen técnicas de programación avanzadas para el acceso secuencial eficiente a ficheros directos.
- Relacionado con la anterior, pueden quedar muchos huecos libres en el dispositivo de memoria secundaria, desaprovechándose el espacio.

2.4 Archivos de organización relativa aleatoria (o indirecta)

Se denominan así a los **archivos relativos** que empleen alguna **función hash para transformar la clave** y conseguir así la dirección física.

La **función hash** puede ser muy **sencilla**, como la del ejemplo que vimos en el apartado 2.2 (que consistía en multiplicar la clave por 2 para obtener la dirección física) o más **complicada**[2], pero el principio es el mismo: transformar la clave para obtener la dirección física.

Dependiendo de la función hash empleada pueden surgir **colisiones**, es decir, claves que proporcionan **la misma dirección física**.

Por ejemplo, si la función hash es `f(clave) = clave / 2` (división entera), tendremos que los registros con clave 500 y 501 intentarán ocupar la misma dirección física: la 250. Es responsabilidad del programador **evitar estas colisiones** y, en caso de que lleguen a producirse, detectarlas y programar algún mecanismo que las resuelva.

Otras funciones hash, como la ya vista `f(clave) = clave x 2`, no producen colisiones, pero en cambio provocan que muchas direcciones físicas no sean utilizadas, con lo que se **desaprovecha el espacio** de almacenamiento.

Por lo tanto, **la elección de una función hash adecuada es crucial** para el correcto rendimiento y funcionamiento de este tipo de archivos. Existen multitud de funciones hash adaptadas a los más diversos problemas que ofrecen un máximo aprovechamiento del espacio y un mínimo número de colisiones, pero su estudio excede a las posibilidades de nuestra asignatura.

Las **ventajas** de los archivos aleatorios son similares a las de los directos, y entre los **inconvenientes** podemos quitar el de dejar muchos huecos libres, siempre que, como hemos visto, la función hash elegida sea adecuada.

[2] Por ejemplo, he aquí una posible función hash más realista: `f(clave) = clave * num_primo + clave`, donde "num_primo" es el número primo más cercano que exista a 2^n, siendo n el número de bits de la clave.

2.5 Archivos de organización indexada[3]

Se denomina así a una **mezcla entre la organizaciones secuencial y relativa**, que pretende aprovechar las ventajas de las dos organizaciones, evitando al mismo tiempo sus inconvenientes.

Los archivos indexados están divididos en **tres zonas o áreas**:

1. ***El área primaria***. *En esta área se encuentran almacenados los registros del archivo secuencial. Es decir, el área primaria es, en realidad, un archivo secuencial corriente[4]. Los registros deben estar ordenados (normalmente, se hará en orden creciente según sus claves)*

   *El área primaria suele estar **segmentada**, es decir, dividida en trozos o segmentos. En cada segmento se almacenan N registros en posiciones de memoria consecutivas. Para acceder a un registro individual, primero hay que acceder a su segmento y, una vez localizado el segmento, buscar secuencialmente el registro concreto.*

2. ***El área de índices***. *Se trata, en realidad, de un segundo archivo secuencial agregado al primero. Pero es un archivo especial, cuyos registros solo tienen dos campos: uno contiene la clave del último registro de cada segmento, y otro contiene la dirección física de comienzo de cada segmento.*

3. ***El área de excedentes***. *Puede ocurrir que los segmentos del área primaria se llenen y no puedan contener algún registro. Esos registros van a parar a un área de excedentes u overflow.*

Para **acceder a un registro** concreto en un archivo indexado, el procedimiento es el siguiente:

- Primero, **buscamos secuencialmente en el área de índices** la dirección de comienzo del segmento donde está el registro que queremos buscar.
- Segundo, hacemos un **acceso directo al primer registro del segmento**.
- Después hacemos un **recorrido secuencial dentro del segmento** hasta localizar el registro.
- Si el registro no se encuentra, acudimos al **área de excedentes** y hacemos un nuevo recorrido secuencial en ella para intentar localizarlo allí.

Observa que los archivos indexados mezclan los accesos secuenciales con los accesos directos.

Ejemplo de archivo indexado

Vamos a mostrar un ejemplo para tratar de entender correctamente esta organización de archivo.

Supongamos un **archivo** de datos personales de los alumnos que conste de estos **10 registros**:

DNI (clave)	Nombre	Teléfono
1111	Arturo Pérez	348734
1232	Miguel Ruiz	349342
2100	Antonia Camacho	209832
2503	Silvia Ortiz	349843
3330	Sonia del Pino	987349
5362	José Anguita	978438
6300	Ana Zamora	476362
6705	Susana Hernández	473239
7020	Rodrigo Sánchez	634838
9000	Natalia Vázquez	362653

Imaginemos que cada segmento tiene **4 registros**. Por lo tanto, el archivo se dividirá en **3 segmentos**. Si suponemos que cada registro ocupa **50 bytes** en memoria secundaria, y que el principio del archivo está en la **dirección 100** de dicha memoria, el archivo físico tendrá este aspecto:

Área primaria				
Dirección física	Clave (DNI)	Contenido del registro		
100	1111	Arturo Pérez	348734	*Segmento 1*
150	1232	Miguel Ruiz	349342	
200	2100	Antonia Camacho	209832	
250	2503	Silvia Ortiz	349843	
300	3330	Sonia del Pino	987349	*Segmento 2*
350	5362	José Anguita	978438	
400	6300	Ana Zamora	476362	
450	6705	Susana Hernández	473239	

[3] La Real Academia de la Lengua admite tanto la denominación "indexada" (del inglés "index") como la menos habitual "indizada" (del castellano "índice")

[4] También hay archivos indexados cuya área primaria es un archivo directo, pero no los vamos a estudiar en este curso.

500	7020	Rodrigo Sánchez	634838
550	9000	Natalia Vázquez	362653
600	Sin usar		
650	Sin usar		

Segmento 3

Área de índices		
Segmento	Dirección de comienzo	Clave del último registro
1	100	2503
2	300	6705
3	500	9000

Observa primero el **área primaria**: los registros están dispuestos en **orden creciente** según la clave (que, en este caso, es el campo NIF). A la izquierda aparece la dirección física donde comienza cada registro. Fíjate también en que los registros están agrupados en tres segmentos.

Luego fíjate en el **área de índices**: contienen una **lista de segmentos**, guardando la dirección de comienzo del segmento y la clave del último registro de ese segmento.

Para acceder, por ejemplo, al registro cuya clave es **5362**, el proceso es el siguiente:

1. **Buscar en el área de índices secuencialmente**, es decir, desde la primera fila, hasta localizar un registro mayor que el que estamos buscando. Eso ocurre en la segunda fila, pues la clave del último registro es 6705. Por lo tanto, sabemos que el registro buscado debe de estar en el **segmento 2**.
2. **Acceder de forma directa a la dirección 300** del área primaria, que es de comienzo del segmento 2. Esa dirección la conocemos gracias a que está guardada en el área de índices.
3. **Buscar en el área primaria secuencialmente** a partir de la dirección 300, hasta localizar el registro buscado, que ocupa la segunda posición dentro de ese segmento.

Fíjate que han sido necesarios, en total, **4 accesos** secuenciales y **1 directo**. Si hubiésemos hecho una búsqueda secuencial, hubiéramos necesitado **6 accesos** secuenciales desde el principio del archivo. Esto puede no parecer una gran ventaja, pero ahora piensa qué pasaría si el archivo tuviera más segmentos y el registro buscado estuviera muy lejos del principio del archivo. Cuanto mayor es el tamaño del archivo y más lejos del principio está el registro, más ventajosa resulta la organización indexada frente a la secuencial.

3 Los archivos en Java

Hasta ahora hemos visto las formas de organización de archivos. En este apartado y el siguiente vamos a estudiar qué clases nos ofrece el JDK de Java para manejar archivos, y cómo utilizarlas.

3.1 Los flujos de datos

Java utiliza un único objeto, con múltiples variantes, para manejar ficheros: **el flujo o *stream***.

En realidad, Java utiliza flujos para manejar cualquier tipo de entrada o salida de datos de la aplicación, sea cual sea su origen o su destino. Así, la consola (salida) y el teclado (entrada) también se manejan como flujos, y, de hecho, lo has estado usando así hasta ahora sin saberlo. Otras conexiones de datos son asimismo

flujos. Por ejemplo, una salida de datos a través de un puerto de red, o una entrada de datos procedente de otro programa.

El flujo es una forma conveniente de **abstraer los detalles de implementación**, de manera que al programador le resulta más fácil manejar cualquier medio de entrada o salida. Es evidente que no es lo mismo enviar una cadena de caracteres a la pantalla que enviarla por el interfaz de red. Pero si disponemos de dos clases derivadas de Stream que nos proporcionan métodos idénticos para enviar nuestra salida a ambos sitios, y ellas solitas se encargan de traducirlo al "lenguaje" propio de cada uno de los destinatarios, la labor del programador se simplifica enormemente.

Todos los flujos se programan, pues, del mismo modo. Si lo que queremos en **escribir** información en el flujo, haremos lo siguiente:

```
1. Abrir el flujo
2. Mientras exista información para enviar al flujo
   {
        Escribir la información en el flujo
   }
3. Cerrar el flujo
```

Del mismo modo, para **leer** información procedente de un flujo, el algoritmo básico es:

```
1. Abrir el flujo
2. Mientras exista informacion procedente del flujo
   {
       Leer la información del flujo
   }
3. Cerrar el flujo.
```

3.2 Clases para manejar flujos en memoria

En el paquete java.io están todas las clases relativas a flujos de la biblioteca estándar de Java. El paquete contiene interfaces, clases y excepciones relacionadas con la entrada salida.

Insistimos en la idea: los flujos permiten enviar y recibir información desde y hacia cualquier sitio. Verás que las siguientes clases pueden leer y escribir información ya residente en la memoria principal, y no en memoria secundaria. Así que te encontrarás a un flujo leyendo caracteres de un String, por ejemplo, para mostrarlos por pantalla. Puede que te parezca que no tiene mucha utilidad, pero nuestro objetivo ahora es entender bien cómo funcionan los flujos antes de aplicarlos a ficheros de disco.

Básicamente, existen dos conjuntos de clases: las que se utilizan para leer y escribir cadenas de caracteres, y las que se utilizan para leer y escribir información binaria. Insistimos: leer y escribir en *cualquier* dispositivo.

Clases para lectura y escritura de caracteres en memoria principal:

Reader (clase abstracta), de la que derivan, entre otras:

- StringReader: para leer cadenas.
- CharArrayReader: para lee arrays de caracteres.

Write (clase abstracta), de la que derivan, entre otras:

- StringWriter: para escribir cadenas.
- CharArrayWriter: para escribir arrays de caracteres.

Clases para lectura y escritura de información binaria en memoria principal:

InputStream (clase abstracta), de la que derivan, entre otras:

- FileInputStream: para leer de ficheros de disco.
- StringBufferInputStream: para leer la información convertida en cadenas de caracteres.

- ByteArrayInputStream: para leer directamente los bytes del flujo binario.
- PipedInputStream: para leer datos procedentes de otro programa (*pippeling*)

OutputStream (clase abstracta), de la que derivan, entre otras:

- StringBufferOutputStream: para escribir la información binaria a partir de cadenas de caracteres.
- ByteArrayOutputStream: para escribir directamente los bytes binarios en el flujo.
- PipedOutputStream: para escribir en una tubería datos destinados a otro programa (*pippeling*)

Todas estas clases, y otras que existen en java.io, se utilizan del mismo modo según los algoritmos básicos de entrada/salida que vimos más arriba.

Vamos a verlo con un ejemplo. La siguiente clase se encarga de crear un flujo de entrada para leer un string, y un flujo de salida para ir "escribiendo" la información, con la que luego construiremos un array de caracteres para mostrarlo en pantalla.

```
import java.io.*;

public class EjemploFlujos
{
  public static void main(String[] args)
  {
    String s = new String("No por mucho madrugar amanece más temprano\n");
    s = s + "No por mucho madrugar amanece más temprano\n";
    s = s + "No por mucho madrugar amanece más temprano\n";
    s = s + "Una novela de Jack Torrance";

    char[] arr = new char[s.length()];
    int car = 0;

    StringReader flujoEntrada = new StringReader(s);
    CharArrayWriter flujoSalida = new CharArrayWriter();

    try
    {
      while ((car = flujoEntrada.read()) != -1)
      {
        flujoSalida.write(car);
      }
      arr = flujoSalida.toCharArray();
      System.out.println(arr);
    }
    catch (IOException e) { e.printStackTrace(); }
    finally
    {
      flujoEntrada.close();
      flujoSalida.close();
    }
  }
}
```

3.3 Clases para modificar el comportamiento de los flujos

Además de las clases de entrada/salida anteriores, existe otra colección de clases para alterar el comportamiento de los flujos. Por ejemplo, pueden realizar una conversión de tipos, añadir un filtro o crear

un buffer. Algunas de estas clases son:

Para flujos de caracteres:

- **BufferedReader** y **BufferedWriter**: añaden un buffer o memoria temporal al flujo, de modo que los caracteres se van almacenando en el buffer hasta que éste se llena o recibe la orden de vaciado, momento en el cual los caracteres siguen su trayecto. Un ejemplo de flujo con buffer que has usado varias veces hasta hoy sin saberlo es este:

```
br = new BufferedReader(new InputStreamReader(System.in));
cadena = br.readLine();
```

  ¿Te suena? Es la forma estándar de hacer la entrada por teclado. Se crea un flujo de entrada procedente de System.in, y se modifica con BufferedReader para que tenga buffer. Así, el flujo puede almacenar todos los caracteres tecleados hasta que, por ejemplo, se pulsa Intro, momento en el cual el buffer se libera y envía sus datos al receptor.
- **InputStreamReader** y **OutputStreamWriter**: transforman flujos de bytes en flujos de caracteres. Por eso es la otra clase utilizada en la entrada por teclado. Los teclados envían códigos binarios al puerto de entrada del ordenador, y éstos se transforman en caracteres dentro del flujo.
- **FilterReader** y **FilterWriter**. Permiten aplicar filtros al flujo de datos.
- **PushbackReader**. Puede mirar cuál es el siguiente carácter del flujo antes de leerlo efectivamente y rechazarlo si no es aceptado (unread)
- **PrintWriter**. Permite escribir las variables de Java con apariencia "normal". Se usa para la depuración.

Para flujos binarios:

- **BufferedInputStream** y **BufferedOutputStream**: añaden un buffer al flujo.
- **ObjectInputStream** y **ObjectOutputStream**: serializan y deserializan un objeto. La serialización es un proceso necesario para poder enviar un objeto en forma binaria a otro dispositivo (por ejemplo, para almacenarlo en una base de datos)
- **FilterInputStream** y **FilterOutputStream**. Permiten aplicar filtros al flujo de datos.
- **DataInputStream** y **DataOutputStream**. Permiten modificar datos desde y hacia los formatos propios de Java procedentes o destinados a otras arquitecturas.
- **PushbackInputStream**. Puede mirar cuál es el siguiente carácter del flujo antes de leerlo efectivamente y rechazarlo si no es aceptado (unread)
- **PrintStream**. Permite escribir las variables de Java con apariencia "normal". Se usa para la depuración.
- **SequenceInputStream**. Tiene capacidad de concatenar las entradas.

3.4 ¿Con buffer o sin buffer?

Los flujos, como ves, pueden tener o no un **buffer**. Pero, ¿qué demonios es un buffer?

Se puede pensar en un buffer como si fuera un array donde se van almacenando los datos dirigidos al flujo, o los datos que el flujo envía hacia el programa. Esos datos se van colocando en el buffer hasta que éste se llena, y sólo entonces pasan efectivamente a su destinatario. También es posible **forzar el vaciado del buffer** antes de que se llene.

En resumen: cuando se envían datos a través de un flujo con buffer, éstos llegan inmediatamente a su destinatario, sino que se van acumulando en el buffer, y sólo cuando el buffer está lleno los datos se envían realmente. En ese momento el buffer queda vacío y listo para seguir recibiendo datos. El mismo proceso, pero a la inversa, se produce con los flujos de entrada.

En cambio, si el flujo no tiene buffer, los datos enviados al flujo son transmitidos a su destinatario de forma

inmediata. Esto produce una comunicación más fluida pero también requiere el consumo de más recursos, y que el receptor de la información este continuamente procesando la información que le llega del flujo.

La decisión de usar uno u otro tipo de flujo dependerá del tipo de entrada y salida que queramos hacer, y del dispositivo al que estemos enviando o del que estemos recibiendo información. Por ejemplo, **la entrada por teclado siempre se hace con buffer**, porque, de lo contrario, los caracteres llegarían al programa inmediatamente después de la pulsación de cada tecla, y no en cadenas completas enviadas al pulsar Intro. Y la salida por pantalla se hace sin buffer, para que se refleje en la misma en el instante en el que se envían los datos. Si hubiese un buffer, las salidas no aparecerían hasta que éste estuviese lleno.

Como regla general, **siempre que queramos leer de un flujo cadenas de caracteres completas** (hasta el fin de línea o retorno de carro), necesitaremos **usar un buffer**. En caso contrario, tendremos que leer carácter a carácter.

Al cerrar un flujo con buffer, éste se vacía, es decir, se terminan de enviar los últimos datos que pudieran quedar en él. También es posible forzar el vaciado prematuro con un método que suele denominarse *flush()*.

4 Clases para manejar ficheros

¡Por fin! Después de este recorrido por las múltiples clases para manipular flujos, ahora aterrizamos en las clases que nos van a permitir abrir un flujo conectado con un ficheros de disco (o, para ser estrictos, ficheros en memoria secundaria)

Ya habrás deducido que la manipulación de flujos es un asunto complejo que en Sun Microsystems se tomaron muy en serio cuando diseñaron la biblioteca de clases de Java. Una de las críticas que se le hacen a Java es que, en su afán por contemplar todas las posibilidades, los diseñadores de la biblioteca estándar incorporaron *demasiadas* clases a la misma. Tal vez ahora empieces a estar de acuerdo con esta crítica.

Las clases que permiten manejar flujos con ficheros no son una excepción. Podemos distinguir, al menos, cuatro grupos de clases. Los clasificamos a continuación para que tengas un mapa del territorio. Después, las describiremos de una en una.

- **Clase genérica**: File
- **E/S binaria**: FileOutputStream y FileInputStream
- **E/S de texto**: FileWriter y FileReader
- **E/S de tipos primitivos**: DataOutputStream y DataInputStream
- **E/S de objetos** (serialización): ObjectOutputStream y ObjectInputStream.

4.1 La clase File

La clase File es la manera más útil para trabajar con ficheros, pero no está diseñada para leer y escribir información en ellos. ¿Para qué sirve, entonces? Calma, muy pronto lo verás.

La clase File tiene métodos para realizar **cualquier operación con el fichero, pero no con su contenido**. ¿Recuerdas que, al principio de este tema, hablábamos de dos tipos de operaciones? Estaban las operaciones con el fichero en su conjunto (crear, eliminar, renombrar, etc), y las operaciones con los datos del interior del fichero. Pues bien, la clase File está pensada para el primer tipo de operaciones.

Tiene, como es costumbre, el **constructor** sobrecargado. Aquí tienes los constructores más útiles:

- File(String rutaAbsoluta): abre un flujo a un fichero. Hay que especificar la ruta completa, incluyendo el nombre del fichero.
- File(String ruta, String nombreFichero): abre un flujo a un fichero. En el primer argumento se especifica la ruta de directorios y, en el segundo, el nombre del fichero.

A continuación te presentamos algunos de los **métodos** de la clase File para **manejar ficheros**:

- **isDirectory**(): devuelve true si el fichero es un directorio.
- **isFile**(): devuelve true si el fichero es un fichero de datos.
- **exists**(): devuelve true si el fichero existe.
- **createNewFile()**: si el fichero no existe, lo crea. Observa que es perfectamente posible abrir un flujo a un fichero inexistente para crearlo después.
- **delete**(): borra el fichero. Devuelve true si lo consigue y false si se produce un error al borrar.
- **renameTo**(): cambia el nombre del fichero. También devuelve true o false.
- **canRead**(): devuelve true si el fichero tiene permisos de lectura.
- **canWrite**(): devuelve true si el fichero tiene permisos de escritura.
- **getPath**(): devuelve la ruta al archivo en un String.
- **getAbsolutePath**(): devuelve la ruta completa al archivo, incluyendo el nombre del mismo.
- **getName**(): devuelve el nombre del archivo.
- **getParent**(): devuelve el nombre del directorio padre.
- **length**(): devuelve el tamaño del archivo en bytes.

También hay algunos métodos para **manejar directorios**:

- **mkdir**(): crea un directorio. Devuelve true si lo consigue.
- **list**(): devuelve un array String[] con los nombres de los ficheros del directorio actual.
- **list**(String filtro): devuelve un array String[] con los ficheros que cumplen el filtro (por ejemplo, "*.txt")
- **listFiles**(): devuelve un array File[] con los ficheros del directorio actual.

Aquí tienes un ejemplo sencillo que ilustra el uso de algunos de estos métodos.

```
public class FileDemo
{
   public static void main(String[] args)
   {
      File f = null;
      String[] strs = {"test1.txt", "test2.txt"};
      try{
         // Para cada String en el array strs...
         for(String s:strs )
         {
            // Crea el flujo
            f= new File(s);

            // Si el fichero no existe, lo creamos
            if (!f.exists())
               f.createNewFile();

            // Comprueba si el fichero tiene permisos de escritura
            boolean w = f.canWrite();

            // Recupera la ruta absoluta
            String p = f.getAbsolutePath();
```

```
            // Muestra los datos recuperados
            System.out.print("Ruta: " + p + ". Permiso escritura: " + w);
         }
      }
      catch(Exception e) { e.printStackTrace(); }
   }
}
```

4.2 E/S binaria

Existe una clase derivada de OutputStream que se llama **FileOutputStream**, y que nos permitirá escribir información binaria en un fichero, es decir, bytes con cualquier información codificada en ellos. Análogamente, hay una clase derivada de InputStream llamada **FileInputStream** que nos permitirá leer información binaria de un fichero.

4.2.1 FileOutputStream

Los **constructores** de FileOutputStream son:

- FileOutputStream(String nombreFichero): abre un flujo de salida hacia el fichero indicado. Si el fichero ya existe, lo borra. Si no existe, se produce una excepción.
- FileOutputStream(File fichero): como el anterior, pero abriendo el flujo a partir de un objeto File.
- FileOutputStream(String nombreFichero, boolean añadir): abre un flujo de salida hacia el fichero indicado. Si el fichero ya existe, añade información a continuación del último byte (es decir, no lo borra). Si no existe, se produce una excepción.
- FileOutputStream(File fichero, boolean añadir): como el anterior, pero abriendo el flujo a partir de un objeto File.

Se trata de una clase bastante simple. Sus **métodos** más usuales son:

- **write**(byte b): escribe el byte b en el flujo, a continuación del último byte escrito. Apto para el acceso secuencial.
- **write**(byte[] b, int offset, int cant): escribe cant bytes en el flujo, comenzando en la posición offset (desplazamiento calculado desde el principio). Apropiado para el acceso directo y aleatorio.
- **close**(): cierra el flujo.

A continuación, ilustramos el uso de FileOutputStream con un ejemplo:

```
import java.io.*;

public class EjemploFileOutputStream
{
  public static void main(String[] args)
  {
    FileOutputStream fop = null;
    File file;
    String content = "Esto es lo que escribiremos en el fichero";
    try
    {
      file = new File("mi_fichero.txt");
      fop = new FileOutputStream(file);

      // Si el fichero no existe, lo creamos
      if (!file.exists())
      {
        file.createNewFile();
      }
```

```
      // Convertirmos el string en bytes para poder escribirlo
      // (FileOutputStream solo admite bytes)
      byte[] contentInBytes = content.getBytes();

      fop.write(contentInBytes);
      fop.flush();
      fop.close();

      System.out.println("Escritura realizada con éxito");
    }
    catch (IOException e) { e.printStackTrace(); }
    finally
    {
      try
      {
        if (fop != null) { fop.close(); }
      }
      catch (IOException e) { e.printStackTrace(); }
    }
  }
}
```

4.2.2 FileInputStream

Para la clase FileInputStream tenemos estos **constructores**:

- FileInputStream(String nombreFichero): abre un flujo de entrada desde el fichero indicado.
- FileInputStream(File fichero): abre un flujo de entrada a partir de un objeto File, que se estudiará más adelante.

Los **métodos** más usuales son:

- **available**(): devuelve el número de bytes que restan por leer en lectura secuencial.
- **read**(): lee un byte del flujo. Apropiado para el acceso secuencia. Devuelve -1 si no hay datos en el flujo.
- **read**(byte[] b, int offset, int cant): lee cant bytes del flujo, comenzando por la posición offset (desplazamiento contado desde el comienzo del flujo). Apropiado para el acceso directo y aleatorio.
- **skip**(int n): salta n bytes adelante en la lectura del flujo.
- **close**(): cierra el flujo.

A continuación, ilustramos la lectura de un fichero mediate FileInputStream con un ejemplo sencillo. Observa como se recorre el flujo con un bucle while cuya condición es que el byte leído del flujo sea distinto de -1. Esto es porque, cuando en el flujo no quedan datos, el método read() devuelve -1. Así sabemos que hemos alcanzado el fin del fichero:

```
public class ReadFileExample
{
  public static void main(String[] args)
  {
    File file = new File("mi_fichero.txt");
    FileInputStream fis = null;
    try
    {
      fis = new FileInputStream(file);
      System.out.println("Tamaño del fichero: " + fis.available());
      int content;
      while ((content = fis.read()) != -1)
      {
        // Convierte el byte leído en un carácter para mostrarlo
```

```
        System.out.print((char) content);
      }
    }
    catch (IOException e) { e.printStackTrace(); }
    finally
    {
      try
      {
        if (fis != null)
          fis.close();
      }
      catch (IOException ex) { ex.printStackTrace(); }
    }
  }
}
```

4.3 E/S de caracteres

Leer y escribir bytes en un fichero es todo lo que cualquier programador necesita para manipular la memoria secundaria, ya que cualquier información puede codificarse, en última instancia, como un flujo de bytes. No importa si es un carácter, una cadena o un objeto completo: al final podrás apañártelas para codificarlo como una secuencia de bytes, escribirlo y después leerlo para recuperarlo.

Pero existen otras clases diseñadas específicamente para leer y escribir información codificada de otros modos habituales. Esto te ahorrará mucho trabajo.

Las primeras de ellas son FileWriter y FileReader. Están pensadas para enviar y recibir caracteres, de modo que son las más cómodas si se trata de manejar ficheros de texto. En los ejemplos anteriores ya hemos visto que se pueden manejar ficheros de texto con FileOutputStream y FileInputStream, pero realmente es mucho más cómodo hacerlo con FileWriter y FileReader.

4.3.1 FileWriter

La clase FileWriter se utiliza para abrir un flujo de salida hacia un fichero de texto. Algunos de sus **constructores** son:

- FileWriter(File f): construye el objeto a partir de otro objeto File. Si el fichero ya existe, lo borra.
- FileWriter(File f, boolean añadir): igual que el anterior, pero permite indicar si queremos añadir datos al final de un fichero ya existente o borrarlo.
- FileWriter(String nombreFichero): construye el objeto a partir del nombre del fichero.
- FileWriter(String nombreFichero, boolean añadir): como el anterior, pero permite indicar si queremos añadir datos al final de un fichero ya existente o borrarlo.

En cuanto a los **métodos**, son similares a los de FileOutputStream, pero modificados para admitir caracteres en lugar de bytes:

- **write**(char c): escribe el carácter c en el flujo, a continuación del último carácter escrito. Apto para el acceso secuencial.
- **write**(char[] c): escribe una cadena de caracteres.
- **write**(String s): escribe el contenido de un String.
- **write**(char[] c, int offset, int cant): escribe cant caracteres en el flujo, comenzando en la posición offset (desplazamiento calculado desde el principio). Apropiado para el acceso directo y aleatorio.
- **write**(String s, int offset, int cant): como el anterior, pero usando un String en lugar de un char[].
- **close**(): cierra el flujo.

Veremos un ejemplo de código que usa FileWriter conjuntamente con el ejemplo para FileReader, en el siguiente apartado.

4.3.2 FileReader

La clase FileReader sirve para manejar flujos de entrada procedentes de ficheros de caracteres.

Sus **constructores** más habituales son:

- FileReader(File f): construye el objeto a partir de otro objeto File. Si el fichero ya existe, lo borra.
- FileReader(String nombreFichero): construye el objeto a partir del nombre del fichero.

Y sus **métodos** no deberían causar ninguna sorpresa a estas alturas:

- **read**(): lee un carácter del flujo, el siguiente del último carácter leído. Apto para el acceso secuencial.
- **read**(char[] c): lee todos los caracteres disponibles en el flujo de entrada y los coloca en el array c. Devuelve el número de caracteres leídos, o -1 si no pudo leer nada.
- **read**(char[] c, int offset, int cant): lee cant caracteres en el flujo, comenzando en la posición offset (desplazamiento calculado desde el principio), y los coloca en el array char[]. Devuelve el número de caracteres leídos, o -1 si no pudo leer nada. Apropiado para el acceso directo y aleatorio.
- **close**(): cierra el flujo.

A continuación mostramos un ejemplo sencillo de uso de las clases FileWriter y FileReader para manejar ficheros de texto. Usaremos un objeto de FileReader para leer, carácter a carácter, un fichero de texto. Iremos almacenando esos caracteres en cadenas de hasta 100 caracteres, y escribiéndolas a través de un FileWriter en otro fichero. Al final, obtendremos una copia del primer fichero.

```
import java.io.*;

public class EjemploESTexto
{
   public static void main(String[] args)
   {
      int c = 0;
      int i = 0;
      char[] cad = new char[100];
      try
      {
         FileReader finput = new FileReader("fichero_origen.txt");
         FileWriter foutput = new FileWriter("fichero_destino.txt");

         // El valor c == -1 indicará fin del fichero de entrada
         while (c != -1)
         {
            i = 0;  // Contador para indexar el array de caracteres
            c = finput.read();
            while ((c != -1) && (i < 100))
            {
               cad[i] = (char)c;  // Guardamos el carácter leído en el array
               i++;
               c = finput.read();
            }
            foutput.write(cad);  // Escribimos en el array
         }
         if (finput != null) finput.close();
         if (foutput != null) foutput.close();
      }
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

4.3.3 FileReader y BufferedReader, una buena pareja

La clase FileReader se usa muy a menudo en combinación con BufferedReader, que permite leer líneas completas hasta el retorno de carro. Observa en este ejemplo como se crea un FileReader que conecta con el fichero de texto, y luego se monta un BufferedReader a partir del FileReader. Las líneas se leen cómodamente con el método readLine() de BufferedReader, hasta alcanzar el fin de fichero.

Esta es la forma habitual de proceder para leer ficheros de texto completos.

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

        // El nombre del fichero que vamos a utilizar
        String fileName = "temp.txt";

        // Esta variable contendrá el texto de cada línea del fichero
        String line = null;

        try {
            FileReader fileReader = new FileReader(fileName);
            BufferedReader bufferedReader = new BufferedReader(fileReader);

            while((line = bufferedReader.readLine()) != null) {
                // Mostramos el contenido del fichero en pantalla
                System.out.println(line);
            }
            bufferedReader.close();
        }
        catch(FileNotFoundException ex) {
            System.out.println("No se puede abrir el archivo");
        }
        catch(IOException ex) {
            System.out.println("Error en la lectura del archivo");

            // Y ya sabes que siempre puedes hacer esto:
            // ex.printStackTrace();
        }
    }
}
```

4.4 E/S de tipos primitivos

A veces necesitamos enviar a un fichero datos de diversa naturaleza: enteros, reales, caracteres... Siempre podemos convertirlos a un formato binario y usar FileOutputStream, o crear un fichero de texto (por ejemplo, XML) para almacenar la información usando FileWriter. Pero también podemos optar por enviarlos directamente a un fichero. La clase DataOutputStream se encargará de convertirlos por nosotros en una secuencia de bytes para que luego FileOutputStream los guarde en un fichero.

Lógicamente, también existe la clase hermana, DataInputStream, para leer los datos escritos y recuperarlos.

4.4.1 DataOutputStream

La clase DataOutputStream crea un flujo de salida para escribir tipos primitivos. Los convierte automáticamente a bytes.

El **constructor** más habitual es:

- DataOutputStream(FileOutputStream f): construye el objeto a partir de otro objeto FileOutputStream.

Sus **métodos** son:

- **writeBoolean**(boolean b): para escribir un dato booleano.
- **writeByte**(byte b): para escribir un dato de tipo byte.
- **writeChar**(char c), **writeShort**(shor s), **writeInt**(int i), **writeLong**(long l), **writeFloat**(float f), **writeDouble**(double d), para escribir datos de cada uno de estos tipos.
- **writeBytes**(byte[] b): para escribir un array de bytes.
- **writeChars**(char[] c): para escribir un array de caracteres.
- **writeUTF8**(chat[] c): para escribir un array de caracteres formateado en UTF8 (Unicode)

Veremos un ejemplo conjunto de uso de DataOutputStream y DataInputStream un poco más abajo.

4.4.2 DataInputStream

La clase DataInputStream crea un flujo de entrada para leer tipos primitivos.

El **constructor** más habitual es:

- DataInputStream(FileInputStream f): construye el objeto a partir de otro objeto FileInputStream.

Sus **métodos** son:

- **readBoolean**(): para leer un dato booleano.
- **readByte**(byte b): para leer un dato de tipo byte.
- **readChar**(), **readShort**(), **readInt**(), **readLong**(), **readFloat**(), **readDouble**(), para leer datos de cada uno de estos tipos.
- **readUTF**(): para leer un array de caracteres formateado en UTF8 (Unicode)

A continuación puedes ver un ejemplo sencillo de uso de DataOutputStream y DataInputStream. Observa como estas clases se encargan de leer y escribir en un flujo, pero necesitan un FileOutputStream y un FileInputStream para que ese flujo comunique con un fichero.

Se trata de un programa que pide los datos (nombre y edad) de un conjunto de alumnos de un instituto y los vuelca en un fichero llamado alumnado.txt. Posteriormente, lee el fichero y muestra la información por pantalla.

Observa, en el último bloque try-catch, como se captura una excepción específica (FileNotFound) para dar un mensaje de error personalizado. Es decir, no todos los catch acaban haciendo aquello de e.printStacktTrace(). Del mismo modo podrías capturar otras excepciones para proporcionar otros mensajes de error, tales como EOFException (EOF = End Of File), CharConversionException, InterruptedByTimeoutException, etc. Como siempre, la lista completa la tienes disponible en docs.oracle.com.

```
import java.io.*;

public class EjemploESTiposPrimitivos
{
   private static FileOutputStream fOutput;
   private static FileInputStream fInput;
   private static DataOutputStream datosOutput;
   private static DataInputStream datosInput;
   private static File f;

   public static void main(String[] args)
   {
      // Escritura del fichero
      try
      {
         fOutput = new FileOutputStream("alumnado.txt");
         datosOutput = new DataOutputStream(fOutput);
         for (int i=0; i<2; i++)
         {
```

```
            System.out.println("Alumno/a núm. " + i);
            String nombre = System.console().readLine("Nombre? ");
                int edad = Integer.parseInt(System.console().readLine("Edad?
"));
            datosOutput.writeUTF(nombre);
            datosOutput.writeInt(edad);
         }
         if (datosOutput != null)
         {
            datosOutput.close();
            fOutput.close();
         }
      }
      catch (IOException e) { e.printStackTrace(); }

      // Lectura del fichero
      try
      {
         f  = new File("alumnado.txt");
         if (f.exists())
         {
            fInput = new FileInputStream(f);
            datosInput = new DataInputStream(fInput);
            String nombre;
            int edad;
            System.out.println("Contenido del fichero alumnado.txt:");
            while (fInput.available() > 0)
            {
               nombre = datosInput.readUTF();
               System.out.println("Nombre: " + nombre);
               edad = datosInput.readInt();
               System.out.println("Edad: " + edad);
            }
         }
         if (datosInput != null)
         {
            datosInput.close();
            fInput.close();
         }
      }
      catch (FileNotFoundException e)
      {
         System.out.println("Fichero alumnado.txt no encontrado");
      }
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

4.4.3 Una nota importante sobre usar excepciones de forma poco excepcional

En el ejemplo anterior ocurre algo importante: observa cómo se construye el bucle que recorre el fichero para leerlo.

```
            while (finput.available() > 0)
            {
               nombre = datos.readUTF();
               System.out.println("Nombre: " + datos.nombre);
               edad = datos.readInt();
               System.out.println("Edad: " + datos.edad);
            }
```

El objeto ObjectInputStream no nos dirá cuántos objetos quedan en el flujo, pero sí que lo hará el FileInputStream con el método available().

Pero, ¡ojo! Algunos, muchos, ¡demasiados! programadores plantearían este bucle así:

```
while (true)
{
   nombre = datos.readUTF();
   System.out.println("Nombre: " + datos.nombre);
   edad = datos.readInt();
   System.out.println("Edad: " + datos.edad);
}
catch (EOFExcepcion e)
{
   System.out.println("Fichero procesado con éxito");
}
```

Es decir, leerían todos los datos del flujo sin medida (while(true)), y dejarían que se provocase una excepción EOF (End Of File) cuando ya no quedasen más datos que leer.

Esta es **una forma incorrecta y bastante perversa de usar las excepciones**, por dos motivos:

- Las excepciones deben reservarse para casos excepcionales. De ahí su nombre. Solo para errores graves y excepcionales, por favor. La ejecución normal de un algoritmo jamás debería pasar por una excepción, o el diseño de algoritmos se convertirá en un infierno peor que el código espagueti de los antiguos GoTos[5].
- Usada así, la excepción EOF no nos sirve para distinguir un fin de fichero normal de un fin de fichero prematuro. Es decir, una excepción EOF nos puede indicar que un fichero está corrupto, pero no si la usamos como mecanismo de salida de un bucle no nos permitirá distinguir un caso del otro.

4.5 E/S de objetos: serialización

La serialización (o seriación) de objetos consiste en transformar su estado en una secuencia de bytes que puedan enviarse a un flujo (típicamente, a un fichero). Esto permite guardar el estado de un objeto, es decir, el valor de sus atributos, en un fichero de memoria secundaria, y reconstruir el objeto más tarde para dejarlo exactamente en el mismo estado en el que estaba[6].

Para poder serializar un objeto, es necesario que su clase implemente el interfaz java.io.Serializable, de este modo:

```
public class Test implements java.io.Serializable
```

Para enviar el estado del objeto a un flujo y para recuperarlo se usan las clases ObjectOutputStream y ObjectInputStream, que describimos a continuación.

4.5.1 ObjectOutputStream

El objeto ObjectOutputStream se **construye** a partir de un FileOutputStream, ya que lo utiliza para conectar con un fichero:

- ObjectOutputStream(FileOutputStream f): construye el objeto a partir de un FileOutputStream.

Utilizaremos básicamente este **método**:

- **writeObject**(objeto): envía al flujo el objeto indicado, que debe implementar el interfaz java.io.Serializable.

[5] La instrucción GoTo de salto incondicional fue la principal causante de la crisis del software de los años 1970 y responsable del ingreso en instituciones psiquiátricas de un buen número de programadores a lo largo de la historia de la informática. Esto es una broma. O no.

[6] Es posible impedir la serialización de ciertos atributos añadiendo el modificador trasient a la declaración. Esto se hace con aquellos atributos que no tienen importancia de cara al estado del objeto y que, por tanto, no es necesario que participen en la serialización. Por ejemplo: public trasient int dato;

4.5.2 ObjectInputStream

El objeto ObjectInputStream se **construye** a partir de un FileInputStream preexistente, a semejanza del objeto de salida:

- ObjectInputStream(FileInputStream f): construye el objeto a partir de un FileInputStream.

Y el **método** que usaremos será éste:

- **readObject**(): devuelve un objeto leído del flujo. Es necesario usar casting para hacer la asignación (ver ejemplo)

Veamos un sencillo ejemplo, en el que varios objetos de la clase Persona (que implementará el interfaz java.io.Serializable) se van a enviar a un fichero y luego a recuperar.

```
public class Persona implements java.io.Serializable
{
   protected String nombre;
   protected int edad;
   public Persona(String n, int e)
   {
      nombre = n;
      edad = e;
   }
   public mostrar()
   {
      System.out.println("Nombre " + nombre, "edad " + edad);
   }
}
```

```
import java.io.*;

public class EjemploESObjetos
{
   static File f;
   static FileOutputStream fo;
   static ObjectOutputStream oos;
   static FileInputStream fi;
   static ObjectInputStream ois;

   public static void main(String[] args)
   {
      String[] nombres = {"Pepe Tardo", "Ana Bolizante", "Jaime Tralleta"};
      int[] edades = {15, 29, 47};

      // Escritura del fichero
      try
      {
         fo = new FileOutputStream("personas.txt");
         oos = new ObjectOutputStream(fo);
         for (int i = 0; i < 3; i++)
         {
            persona p = new persona(nombres[i], edades[i]);
            oos.writeObject(p);
         }
         if (oos != null)
         {
            oos.close();
            fo.close();
         }
      }
      catch (IOException e) { e.printStackTrace(); }
```

```
      // Lectura del fichero
      try
      {
         f = new File("personas.txt");
         if (f.exists())
         {
            fi = new FileInputStream(f);
            ois = new ObjectInputStream(fi);
            persona p = null;

            while (fi.available() > 0)
            {
               p = (persona)ois.readObject();
               p.mostrar();
            }
         }
      }
      catch (IOException e) { e.printStackTrace(); }
      catch (ClassNotFoundException cnf) { cnf.printStackTrace(); }
   }
}
```

4.6 Ficheros de acceso directo aleatorio

(El texto de este apartado es un extracto de *puntocomnoesunlenguaje.blogspot.com.es,* escrito por Enrique García Hernández y publicado originalmente bajo licencia Reconocimiento-NoComercial-CompartirIgual 3.0)

Para hacer **acceso aleatorio** a los ficheros en Java se utiliza la clase **RandomAccessFile.** Los constructores de la clase son:

- **RandomAccessFile**(String path, String modo);
- **RandomAccessFile**(File objetoFile, String modo);

El argumento **modo** indica el modo de acceso en el que se abre el fichero. Algunos de los valores permitidos para este parámetro son:

- **"r"**: Abre el fichero en modo solo lectura. El fichero debe existir.
- **"rw"**: Abre el fichero para lectura y escritura. Si el fichero no existe, lo crea.

Para acceder de forma aleatoria a los datos contenidos en el fichero, la clase RandomAccessFile dispone de varios **métodos**, como, por ejemplo:

- **long getFilePointer()**: Devuelve la posición actual del puntero del fichero. Indica la posición (en bytes) donde se va a leer o escribir.
- **long length()**: Devuelve la longitud del fichero en bytes.
- **void seek(long pos)**: Coloca el puntero del fichero en la posición indicada, medida como desplazamiento en bytes desde el comienzo del fichero. La posición 0 es el principio del fichero, y la posición length() es el final.
- **int read()**: Devuelve el byte leído en la posición marcada por el cursor de lectura. Devuelve -1 si alcanza el final del fichero.
- **String readLine()**: Devuelve una cadena de caracteres leída desde la posición actual del cursor hasta el próximo salto de línea.
- **xxx readXxx()**: Exiten versiones del método read para cada tipo de dato primitivo: readChar, readInt, readDouble, readBoolean, etc.
- **void write(int b)**: Escribe en el fichero un byte.

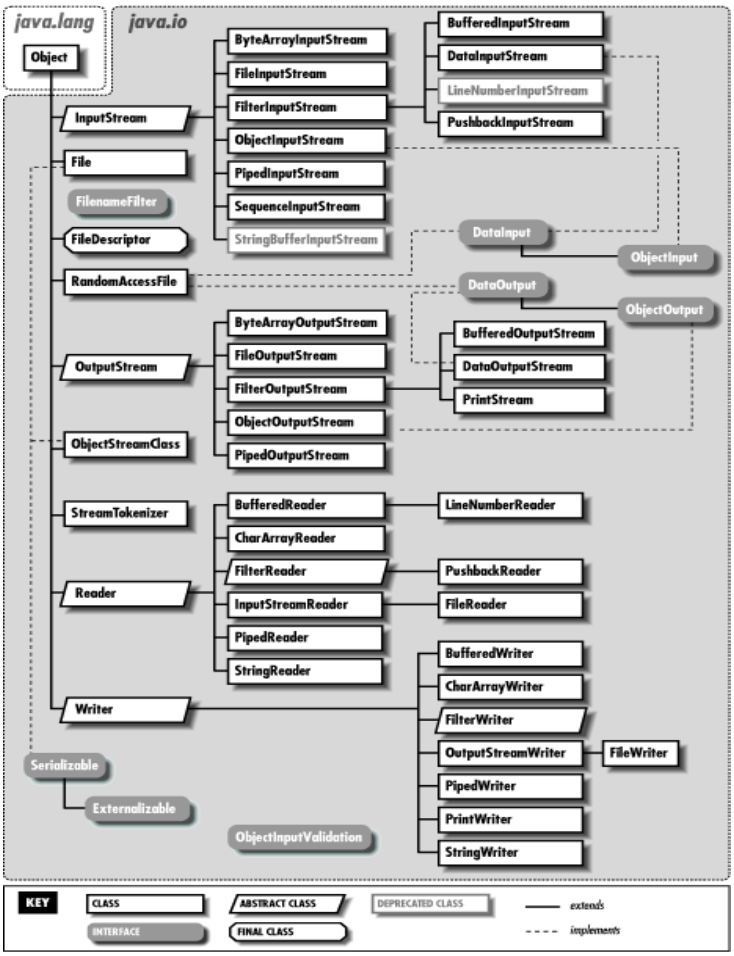
- **void writeBytes(String s)**: Escribe en el fichero una cadena de caracteres.
- **void writeXxx(argumento)**: También existe un método write para cada tipo de dato primitivo: writeChar, writeInt, writeDouble, writeBoolean, etc.

Por ejemplo, el siguiente programa pide un número entero por teclado y lo añade al final de un fichero binario que contiene números enteros. El programa utiliza un método mostrarFichero() que se llama dos veces. La primera muestra el contenido del fichero antes de añadir el nuevo número y la segunda llamada muestra el fichero después de añadirlo.

```
public class FicheroDeEnteros {

    static RandomAccessFile fichero = null;

    public static void main(String[] args) {
        int numero;
        try {
            // Abrimos el fichero para lectura y escritura
            fichero = new RandomAccessFile("numeros.dat", "rw");
            mostrarFichero();
            System.out.print("Introduzca un número entero: ");
            numero = Integer.parseInt(System.console().readLine());
            // Nos situamos al final del fichero
            fichero.seek(fichero.length());
            fichero.writeInt(numero);       // Escribimos el entero
            mostrarFichero();
            fichero.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }

    public static void mostrarFichero() {
        int n;
        try {
            fichero.seek(0); // Nos situamos al principio del fichero
            while (true) {
                n = fichero.readInt();  // Leemos un entero del fichero
                System.out.println(n);
            }
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

4.7 La familia al completo

Ya te hemos presentado a muchas clases de java.io. Déjanos que ahora te mostremos el árbol genealógico completo. No es necesario que conozcas, ni que uses, todas las clases, pero ayuda bastante saber qué tipo de cosas ya están resueltas en la biblioteca estándar, y también en qué rama de la familia se ubica cada una.



5 Procesamiento en Java de archivos secuenciales

En este apartado vamos a estudiar la **implementación en Java** de los algoritmos que habitualmente se utilizan para procesar los archivos secuenciales.

5.1 Escritura

Los registros, en un archivo secuencial, **se añaden siempre al final**. Es necesario crear el flujo con el constructor que permita añadir al final. Si queremos destruir el contenido del archivo, podemos usar el constructor convencional.

Una vez hecho esto, usaremos **sucesivas llamadas al método de escritura** para insertar los registros (si el archivo es binario) o los caracteres (si es de texto). Ten en cuenta que los datos **se grabarán en el archivo exactamente en el mismo orden en el que los escribas**.

Las clases que debes usar dependen de la naturaleza del problema y de las preferencias del programador, pero recuerda que, en general:

- **FileOutputStream** y **FileInputStream** sirven para E/S **binaria** sencilla (flujos de bytes). Si deseas escribir registros complejos, puedes usarla también, formateando los registros como colecciones de bytes, pero es más cómo optar por usarlas conjuntamente con **DataOutputStream** y **DataInputStream** (E/S de tipos primitivos) o con **ObjectOutputStream** y **ObjectInputStream** (serialización de objetos)
- **FileWriter** y **FileReader** sirven para E/S de caracteres.
- Finalmente, la clase **File** sirve de apoyo a todas ellas (todos los flujos pueden crearse a partir de un File), y para operaciones con ficheros y directorios (crear, borrar, renombrar, etc)

Para escribir los datos usaremos siempre el método write() en su versión sencilla, por ejemplo:

- **write**(byte b) (FileOutputStream)
- **write**(String s) (FileWriter)

Estos métodos añaden la información a continuación del último dato escrito, es decir, en forma secuencial.

5.2 Lectura

Cuando se abre un archivo secuencial, el indicador de posición se sitúa **en el primer byte** del archivo. **Cada vez que se lea un dato, el indicador de posición se desplaza** automáticamente tantos bytes adelante como se hayan leído. Las lecturas se pueden continuar haciendo **hasta que se alcance el final del archivo**.

La lectura de archivos secuenciales siempre se plantea como un bucle en el que la condición de salida es haber alcanzado el fin de fichero. Para comprobar esto podemos usar el método available(), que nos dirá cuántos bytes restan por leer en el fichero, o comprobar el valor devuelto por read() (cuando este método esté disponible), que será -1 en caso de haber llegado al EOF, como en este ejemplo:

```
fis = new FileInputStream(file);
byte content;
while (((content = fis.read()) != -1)) && (encontrado == false))
{
  System.out.println((char)content);
}
```

5.3 Búsqueda

En un archivo secuencial **el único método de búsqueda posible es el secuencial**, es decir, que hay que **leer todos los registros**, partiendo del primero, hasta encontrar el que buscamos.

En el siguiente fragmento recorreremos un archivo de forma secuencial buscando el carácter X mayúscula. Si lo encontramos, diremos en qué posición del archivo se encuentra su primera ocurrencia y terminaremos.

Observa que, cuando el carácter no se encuentra, **es necesario recorrer todo el archivo** antes de determinar que el dato no está en el mismo. Este es un problema de la búsqueda secuencial.

Si el archivo estuviera **ordenado** podríamos **mejorar el mecanismo de búsqueda**, ya que no sería necesario recorrer todo el archivo para determinar que un elemento no está: bastaría con encontrar un elemento mayor para poder detener la búsqueda en ese instante. Con los archivos de texto, el concepto de ordenación no tiene mucho sentido, pero con archivos que almacenen datos binarios (por ejemplo, datos del alumnado del instituto ordenados por número de expediente, donde estemos buscando por número de expediente) esto tiene mucho sentido y hace que la búsqueda sea, en promedio, mucho más rápida.

```
fis = new FileInputStream(file);
byte content;
int pos = 0;
boolean encontrado = false;
```

```
      while (((content = fis.read()) != -1)) && (encontrado == false))
      {
        if ((char)content == "X")
        {
           boolean encontrado = true;
        }
        pos++;
      }
      if (encontrado)
      {
        System.out.println("Hemos encontrado el carácter X");
        System.out.println("Posición: " + pos);
      }
      else
        System.out.println("El carácter X no está en este fichero");
```

5.4 Borrado

El borrado (de registros dentro del fichero, no de ficheros completos) es una **operación problemática** en ficheros secuenciales. Existen **dos formas** de hacer el borrado en un archivo secuencial:

- **Crear un segundo archivo** secuencial y **copiar en él todos los registros** del archivo original **excepto el que se pretende borrar**. Después, se borra el archivo original y se renombra el archivo nuevo con el nombre que tenía el original

  Como puedes imaginarte, este método, aunque funciona, **es muy lento**, sobre todo si el archivo es largo.
- **Marcar el registro que se prentende borrar como "no válido"** y, aunque siga existiendo, ignorarlo a la hora de procesar el archivo. Este segundo método requiere acceso directo, así como utilizar registros de estructura compleja (no simples archivos de texto como los que estamos viendo), y se hablará de ellos más adelante y en las actividades del tema.

5.5 Modificación

En los archivos secuenciales sólo puede escribirse al final del archivo. Por lo tanto, para modificar un registro hay que actuar de forma similar al primer método de borrado: **creando un segundo archivo** en el que se **copiarán todos los registros** exactamente igual que en el archivo original, **excepto el que se pretende cambiar**.

Para hacer una modificación directamente sobre los datos existentes necesitamos acceso directo y no solo secuencial.

5.6 Procesamiento de archivos con registros complejos

Los registros complejos son aquellos que contienen varios datos en cada uno de ellos. En un archivo de texto, cada registro es un único carácter. En un archivo binario, en cambio, los bytes no suelen seguir una secuencia azarosa, sino que se organizan en campos que se repiten registro tras registro, dando estructura al archivo.

Por ejemplo, si queremos guardar en un archivo la información pesonal (digamos nombre, edad y número de expediente) de los alumnos de un instituto, lo lógico es que destinemos una cantidad fija de espacio a cada uno de estos datos (digamos 100 bytes para el nombre, 2 bytes para la edad y 4 bytes para el expediente, es decir, char[100], short e int, en terminología de Java)

Esto quiere decir que raramente usaremos un FileOutputStream o un FileInputStream para escribir o leer secuencias de byte sin estructura, sino que generalmente lo montaremos sobre un DataOutputStream o un

ObjectOutputStream. En Java, ambos mecanismos son válidos para leer y escribir archivos binarios con registros complejos.

En caso de usar **DataOutputStream / DataInputStream**, necesitaríamos disponer de tantas variables (y de los tipos adecuados) como tenga cada registro. Por ejemplo:

```
  char[] nombre = new char[100];
  short edad;
  int expediente;
```

Necesitaremos crear el objeto de tipo DataOutputStream o DataInputStream sobre un FileOutputStream o FileInputStream (consulta los apartados correspondientes más arriba si tienes dudas sobre cómo hacerlo), y, utilizando esas variables como fuente o destino de la información del fichero, resulta muy fácil gestionar archivos complejos.

En caso de usar **ObjectOutputStream / ObjectInputStream**, definiremos una clase que implemente el interfaz java.io.Serializable, para poder serializar las instancias de la misma. Esta clase contendrá, como variables miembro, todos los campos de cada registro del fichero. Algo así:

```
class Alumno implements java.io.Serializable {
   char[] nombre;
   short edad;
   int expediente;
   // A continuación, irían los getters y setters para estos atributos
}
```

Como en el caso anterior, el objeto de tipo ObjectOutputStream u ObjectInputStream se crea a partir de un FileOutputStream o FileInputStream. Se crea entonces una instancia de Alumno y se le asignan valores mediante los métodos setters antes de serializarlo para enviarlo al fichero. Consulta más arriba el apartado correspondiente a estas clases si tienes dudas sobre cómo se realiza este proceso.

6 Procesamiento en Java de archivos directos

Recuerda que en los archivos directos el campo clave coincide con la dirección de memoria secundaria donde se almacena el registro (repasa el apartado 2.3 si no sabes de qué estamos hablando)

A la hora de implementar estos archivos **en Java no usaremos realmente la direcciones absolutas de almacenamiento, sino la posición relativa de cada registro respecto del comienzo del archivo**. El primer registro tendrá la **posición 0**, el segundo la posición 1, el tercero la 2, etc. Como los métodos write() nos permite colocarnos al comienzo de cualquier registro, podremos usar los archivos directos como si fueran vectores sin necesidad de conocer la dirección física de almacenamiento.

Lectura y escritura

El método write() tiene versiones que permiten indicar un desplazamiento sobre la posición actual de escritura:

```
write(char[] c, int offset, int cant)
```

El offset (desplazamiento) indica cuántos bytes hay que adelantar la posición de escritura antes de realizarla

efectivamente, contando desde el principio del archivo (posición 0).

De forma análoga, el método read() tiene versiones de este tipo:

```
read(byte[] b, int offset, int cant)
```

Por lo tanto, conocida la posición actual en el archivo, es posible leer o escribir en cualquier otra posición del mismo.

Todo esto es válido tanto para los objetos de las clases FileOutputStream / FileInputStream (archivos binarios) como los FileWriter / FileReader (archivos de texto).

Por ejemplo, si tenemos abierto un fichero de texto para lectura con FileReader y queremos acceder al carácter situado en la posición 100 del mismo, bastará con hacer esto:

```
char c[] = new char[20];  // La longitud del array tiene que ser mayor que 1
f.read(c, 100, 1);        // f es el FileReader
```

Búsqueda

Si los registros **no están ordenados** hay que hacer una **búsqueda secuencial**.

Si el archivo tiene los **registros ordenados** por algún campo tenemos dos opciones:

- Realizar una **búsqueda secuencial mejorada**, de las que se detienen en cuanto encuentran un elemento mayor que el que estábamos buscando
- Realizar una **búsqueda binaria** como la que estudiamos en el tema anterior (en los ejercicios sobre vectores). Se puede aplicar fácilmente a los archivos directos si identificamos cada registro del archivo con una posición del vector. Es un ejercicio muy interesante que plantearemos entre las actividades del tema.

Borrado

Borrar registros puede ser complicado, ya que habría que desplazar el resto del archivo hacia atrás para sobreescribir la posición del registro desaparecido.

Como en el caso de los archivos secuenciales, disponemos de dos métodos para hacerlo:

- **Copiar** todos los registros en otro **archivo auxiliar**, excepto el que se desea borrar.
- **Marcar el registro** que se va a borrar como "no válido".

Como al estudiar los archivos secuenciales nos centramos en el primer método, ahora vamos a referirnos al segundo. Consiste en **marcar los registros** como borrados o no válidos. Para ello, todos los registros deben contar con un campo extra de tipo booleano. Por ejemplo, si hemos optado por manejar los registros con un ObjectStreamOutput, podríamos hacer esto:

```
class Alumno implements java.io.Serializable {
   char[] nombre;
   short edad;
   int expediente;
   boolean borrado;   // Este es el campo que hemos añadido
   // A continuación, irían los getters y setters para estos atributos
}
```

Si el campo borrado vale "false", podemos considerar que el registro es válido. Si vale "true", **el registro se considerará borrado, aunque realmente seguirá existiendo** en el archivo. De este modo, para borrar un registro sólo hay que cambiar el valor de ese campo. **Hemos convertido el borrado en una modificación.**

Si se quiere elaborar más este método, se puede **mantener un fichero auxiliar** con la **lista de los registros borrados**. Esto tiene un doble propósito:

- *Que se pueda diseñar un método write() o read() más eficiente de modo que se tengan en cuenta los registros marcados.*
- *Que al **insertar nuevos** registros, se puedan sobreescribir los anteriormente marcados como borrados, si existe alguno, y así aprovechar el espacio.*

Lo normal es implementar una combinación de los dos métodos de borrado: durante la ejecución normal del programa se borran registros con el método de marcarlos, y cuando se cierra la aplicación, o se detecta que el porcentaje de registros borrados es alto, se **compacta el fichero** usando el segundo método.

Modificación

La modificación consiste en una **sobreescritura de un registro que ya existe**. Al principio de este apartado dedicado a los archivos directos explicábamos como se hacía, al hablar de la lectura y la escritura.

7 Procesamiento en Java de archivos indexados

Lo más ingenioso de los archivos indexados es la **zona de índices**. Repasa el apartado 2.5 si no lo recuerdas con exactitud. Los índices pueden guardarse en **un archivo independiente** o bien en **un array de memoria**. La utilización de arrays hace que los registros se encuentren más rápidamente, pero perderíamos la tabla de índices al cerrar el programa.

Lo más habitual es utilizar una combinación de los dos métodos: manener los índices en un archivo independiente y cargarlo en un array de memoria al inicio del programa, para volver a grabarlos cuando hayamos terminado.

En los siguientes ejemplos supondremos que el área de índices se encuentra en un archivo independiente. El archivo de índices se llamará "índice" y, el de datos, "datos". Qué original.

Búsqueda

Para **leer un registro** primero hay que localizarlo. El procedimiento es el siguiente:

1) **Buscamos secuencialmente en el área de índices** la dirección de comienzo del segmento donde está el registro que queremos buscar. Usaremos la clave del registro buscado para localizar el segmento.
2) Hacemos un **acceso directo al primer registro del segmento**.
3) Hacemos un **recorrido secuencial dentro del segmento** hasta localizar el registro.
4) Si el registro no se encuentra, acudimos al **área de excedentes** y hacemos un nuevo recorrido secuencial en ella para intentar localizarlo allí.

A la hora de implementar todo esto en Java es conveniente **escribir un método** que, recibiendo como parámetro la clave del registro buscado, devuelva su contenido.

Otras operaciones sobre archivos indexados

Manejar archivos indexados es bastante más complejo que hacerlo con archivos secuenciales, pero si se entiende bien el funcionamiento de las búsquedas no será problema implementar el resto de operaciones.

Si hubiera que **insertar datos** nuevos en el archivo, puede hacerse en la zonas vacías del segmento correspondiente (si hay espacio), o puede añadirse un nuevo segmento, siempre que las claves continúen ordenadas. Si hubiera que añadir un registro intermedio, habría que hacerlo en el área de excedentes, que para eso está.

Para **eliminar un registro** basta con borrarlo del área primaria, quedando el hueco vacío con la posibilidad de ser ocupado por otro registro en el futuro, siempre que no se desordenen las claves. Los métodos de borrado son los mismos que en el caso de los archivos secuenciales y directos.

Si se hacen muchas inserciones y borrados, es conveniente **reordenar** el archivo periódicamente, con el fin de **compactar** los segmentos y recolocar los registros que hayan ido a parar al área de excedentes.

El principal problema de la organización indexada es que **el índice puede llegar a ser muy grande** y consumir bastante memoria secundaria. Además, como el índice se recorre de manera secuencial, si es muy largo puede tardarse bastante en llegar a las últimas posiciones. Una solución a este problema es hacer un **índice secundario**, que sirve para indexar la tabla de índices primaria, que a su vez indexa el área primaria del archivo.

Otra posibilidad para reducir el tamaño del índice es **aumentar el tamaño del segmento**, es decir, el número de registros que hay en cada segmento, pero entonces aumentan las búsquedas secuenciales que hay que hacer en el área primaria. Por tanto, hay que **buscar un compromiso** entre el tamaño del segmento y el tamaño del índice para hacer que el número de búsquedas secuenciales sea el mínimo posible.