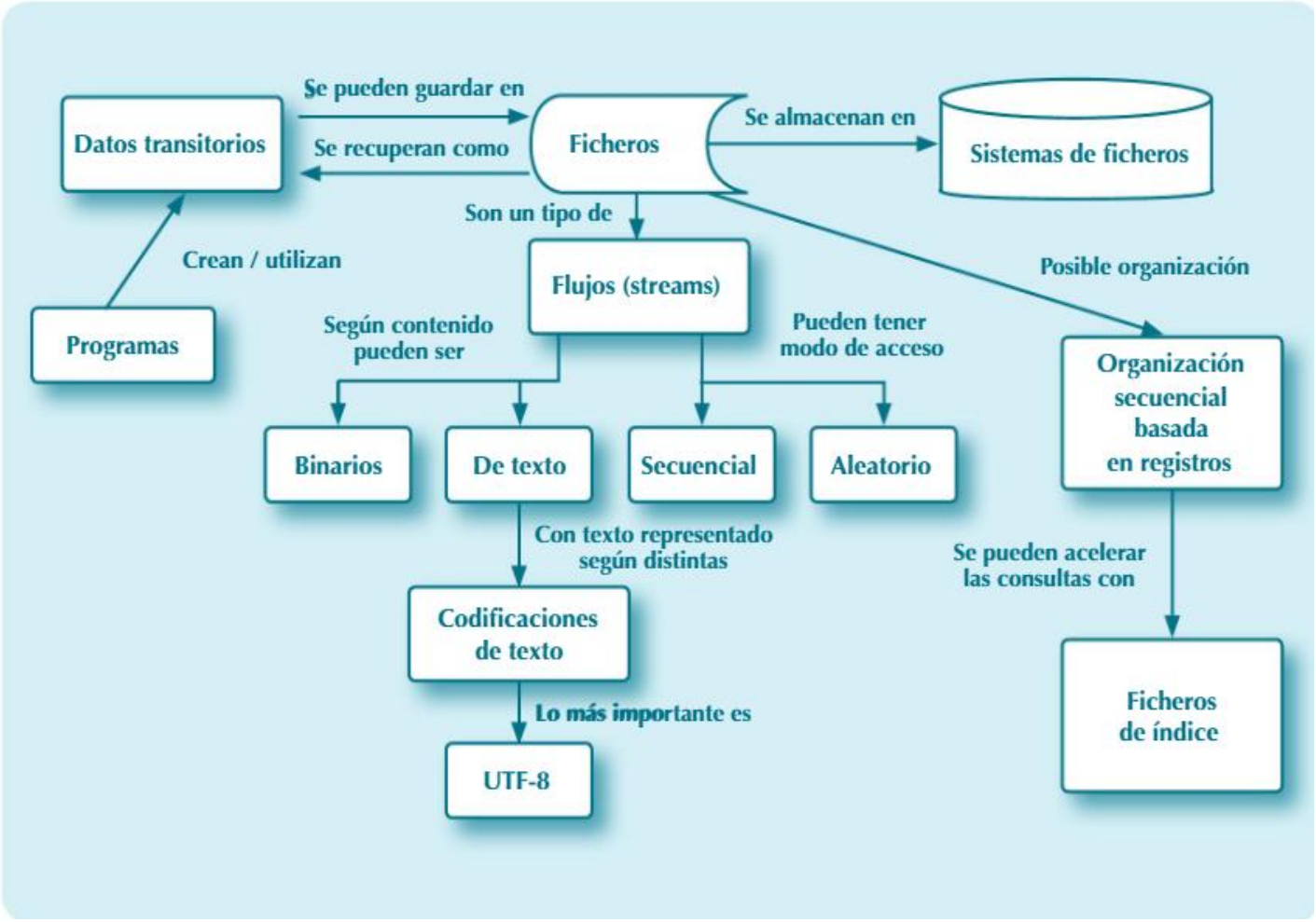
ACCESO A DATOS

FICHEROS

Objetivos

- Conocer las diferencias en la gestión de ficheros de texto y ficheros binarios.
- Diferenciar entre acceso secuencial y aleatorio a ficheros.
- Aprender las principales clases de Java para manejo de ficheros (y flujos en general) y su uso.
- Comprender el mecanismo de *buffering*, cómo permite acelerar las operaciones de lectura y escritura en ficheros tanto binarios como de texto, y cómo permite la lectura y escritura por líneas en ficheros de texto.
- Acceder correctamente a los contenidos de ficheros de texto con distintas codificaciones.
- Analizar algunas organizaciones sencillas de ficheros y la manera en que se pueden utilizar para la persistencia de datos.

Mapa conceptual



Glosario

Acceso aleatorio. Tipo de acceso a un fichero que permite acceder directamente a los datos situados en cualquier posición del fichero.

Acceso secuencial. Tipo de acceso a un fichero con el que la única manera de acceder a los datos situados en una posición determinada es leer todos los contenidos desde el principio hasta dicha posición.

Codificación de texto. Una manera particular de representar una secuencia de caracteres de texto mediante una secuencia de *bytes*.

Fichero de texto. Fichero que contiene texto.

Fichero. Unidad fundamental de almacenamiento. Consiste en una secuencia de *bytes*. Con una adecuada organización, se puede utilizar para almacenar cualquier tipo de información.

Índice. Fichero que permite recorrer los contenidos de otro fichero en un orden determinado.

Registro. Estructura para representar información. Consta de una serie de campos, en cada uno de los cuales se puede almacenar un dato particular.

2.1. Persistencia de datos en ficheros

Los ficheros son el método de almacenamiento de información más elemental. De hecho, en última instancia, todos los métodos de almacenamiento, por sofisticados que sean, almacenan los datos en ficheros.

Hasta que fueron relegados en los años ochenta por las bases de datos relacionales, los ficheros fueron el principal medio de almacenamiento de datos. El nombre **fichero** se utilizó por analogía con los antiguos ficheros que contenían fichas de papel, todas con la misma estructura, consistente en un conjunto fijo de campos. En el caso de los libros de una biblioteca, por ejemplo, los campos podían ser el título, nombre del autor, tema, etc. Los ficheros que contenían fichas de papel fueron reemplazados por ficheros de ordenador que contenían registros, equivalentes a las antiguas fichas de papel. Un registro contiene un conjunto de campos de longitud fija. Para acelerar las búsquedas se empezaron a utilizar ficheros auxiliares de índice que permitían acceder a los registros según un orden determinado. IBM desarrolló un avanzado sistema de gestión de ficheros llamado **ISAM** (*indexed sequential access method*). El lenguaje COBOL, creado en 1959, pero aún hoy ampliamente utilizado en determinados ámbitos (por ejemplo, el sector bancario), proporciona un excelente soporte para ficheros indexados.

Los ficheros como medio de almacenamiento masivo de datos fueron relegados progresivamente en favor de las bases de datos relacionales. En realidad, las bases de datos relacionales utilizan internamente sofisticados sistemas de gestión de ficheros para el almacenamiento de los datos.

En este capítulo se presentarán algunas organizaciones sencillas de ficheros, y se explicará todo lo necesario para escribir sencillos programas en Java para consultar, añadir, borrar y modificar la información contenida en ellos.

TOMA NOTA

El almacenamiento de datos en ficheros de texto con organizaciones sencillas puede ser una solución perfectamente válida para algunas aplicaciones, pero nunca hay que perder de vista sus limitaciones intrínsecas y su limitada escalabilidad. Si hay que realizar consultas complejas o que requiere relacionar mucha información diversa, será difícil escribir un programa para realizarlas. Si el volumen de datos para manejar es muy grande, o si es necesario realizar con mucha frecuencia operaciones de borrado o modificación de datos, el rendimiento será muy pobre. Permitir que varios programas realicen a la vez operaciones de consulta y actualización puede introducir inconsistencias en los datos e incluso dañar los ficheros. Para evitar estos problemas son necesarios elaborados mecanismos de control de acceso que añaden mucha complejidad al sistema, y mucho más complicado es añadir soporte para transacciones. Es difícil establecer y hacer que se cumplan restricciones de integridad sobre los datos. Y también es difícil evitar redundancias en los datos que, además de desperdiciar espacio de almacenamiento, puede hacer que surjan inconsistencias cuando se añaden o modifican datos. Porque si la misma información está en más de un lugar, puede acabar teniendo un valor distinto en cada uno.

Se siguen utilizando ficheros para la persistencia de datos en muchas aplicaciones, y muy importantes. Las bases de datos relacionales han reemplazado los sistemas basados en ficheros para grandes colecciones de datos con una estructura muy regular, lo que es muy importante, pero no lo es todo. Hoy en día se utilizan mucho los ficheros en formato XML (al que se dedicará un capítulo posterior) para el almacenamiento de todo tipo de datos. Los sistemas de correo electrónico suelen mantener sus datos en ficheros con un formato relativamente sencillo. Muchos procesos masivos que se ejecutan periódica o puntualmente se realizan basándose en datos proporcionados en ficheros de texto con una estructura muy sencilla. También se usan ficheros de texto sencillos para exportación e importación de datos entre sistemas, y también para copias de seguridad. Aparte de eso, está la infinidad de aplicaciones que almacenan los datos con los que trabajan en ficheros con infinidad de formatos diferentes.

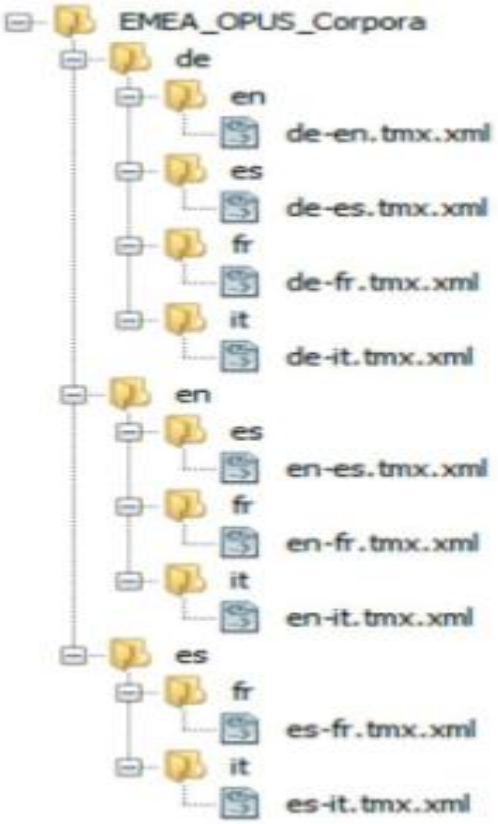
2.2. Tipos de ficheros según su contenido

Un fichero es simplemente una secuencia de *bytes*, con lo que en principio puede almacenar cualquier tipo de información.

Un fichero se identifica por su nombre y su ubicación dentro de una jerarquía de directorios.

Los ficheros pueden contener información de cualquier tipo, pero a grandes rasgos cabe distinguir entre dos tipos: los ficheros de texto y los ficheros binarios:

1. Ficheros de texto: contienen única y exclusivamente una secuencia de caracteres. Estos pueden ser caracteres visibles tales como letras, números, signos de puntuación, etc., y también espacios y separadores tales como tabuladores y retornos de carro. Su contenido se puede visualizar y modificar con cualquier editor de texto, como por ejemplo el bloc de notas en Windows o gedit en Linux.
2. Ficheros binarios: son el resto de los ficheros. Pueden contener cualquier tipo de información. En general, hacen falta programas especiales para mostrar la información que contienen. Los programas también se almacenan en ficheros binarios.



Ficheros en una jerarquía de directorios

2.3. Codificaciones para texto

Aunque la cuestión de las codificaciones para texto se ha incluido en este capítulo dedicado a los ficheros, es relevante para cualquier medio de almacenamiento de datos porque, allá donde se almacene un texto, debe hacerse con una codificación determinada.[1]

Un texto es una secuencia de caracteres. Un texto, como cualquier tipo de información, se almacena en memoria o en cualquier dispositivo de almacenamiento como una secuencia de **bytes**. Una codificación es un método para representar cualquier texto como una secuencia de *bytes*. El mismo texto, según la codificación empleada, se puede representar como una secuencia de *bytes* distinta.

La variedad de codificaciones es un problema cada vez menos importante en la práctica, debido a la implantación general de **Unicode** y su codificación **UTF-8**, pero todavía es relativamente frecuente encontrar textos con otras codificaciones, sobre todo en entornos **Windows**.

Las más frecuentes son **ISO 8859-1** y **Windows-1252**, que se puede considerar una variante no estándar de **Microsoft** del estándar **ISO 8859-1**. **UTF-8** es compatible con el código **ASCII**, lo que significa que cualquier texto codificado en ASCII se representa exactamente igual en UTF-8. Este ha sido un motivo fundamental para la adopción generalizada de UTF-8.

Para las nuevas aplicaciones, siempre hay que utilizar UTF-8 para almacenar texto, a no ser que haya alguna razón de peso para utilizar otra, que normalmente no la hay. A veces hay que importar u obtener textos de otras fuentes, desde donde podrían venir con otras codificaciones. Hay que identificar estas situaciones y hacer la conversión necesaria para recodificar los textos.

*Cualquier editor de texto debería permitir visualizar correctamente un fichero de texto independientemente de su codificación. Por ejemplo, Notepad en **Windows** y **gedit** en **Linux**. Normalmente, con la opción*

*"Guardar como..." se puede seleccionar la codificación que se va a utilizar, y **UTF-8** suele aparecer como una de las opciones.*

*La manera más fiable y segura de confirmar el tipo de un fichero en **Linux** es con el comando **file**. Este analiza los contenidos del fichero e indica su tipo. Si es de texto, indica la codificación utilizada para el texto, típicamente: **ASCII**, **UTF-8** o **iso-8859-1**.*

*El comando **iconv** de **Linux** permite recodificar ficheros de texto. El siguiente comando, por ejemplo, se podría utilizar para recodificar a **UTF-8** un fichero codificado en **ISO 8859-1**.*

```
iconv -f iso8859-1 -t utf-8 fichero_8859-1.txt > fichero_utf8.txt
```

2.3.1 Codificaciones

El siguiente cuadro muestra las principales características de las codificaciones más frecuentes para la codificación del contenido de ficheros.

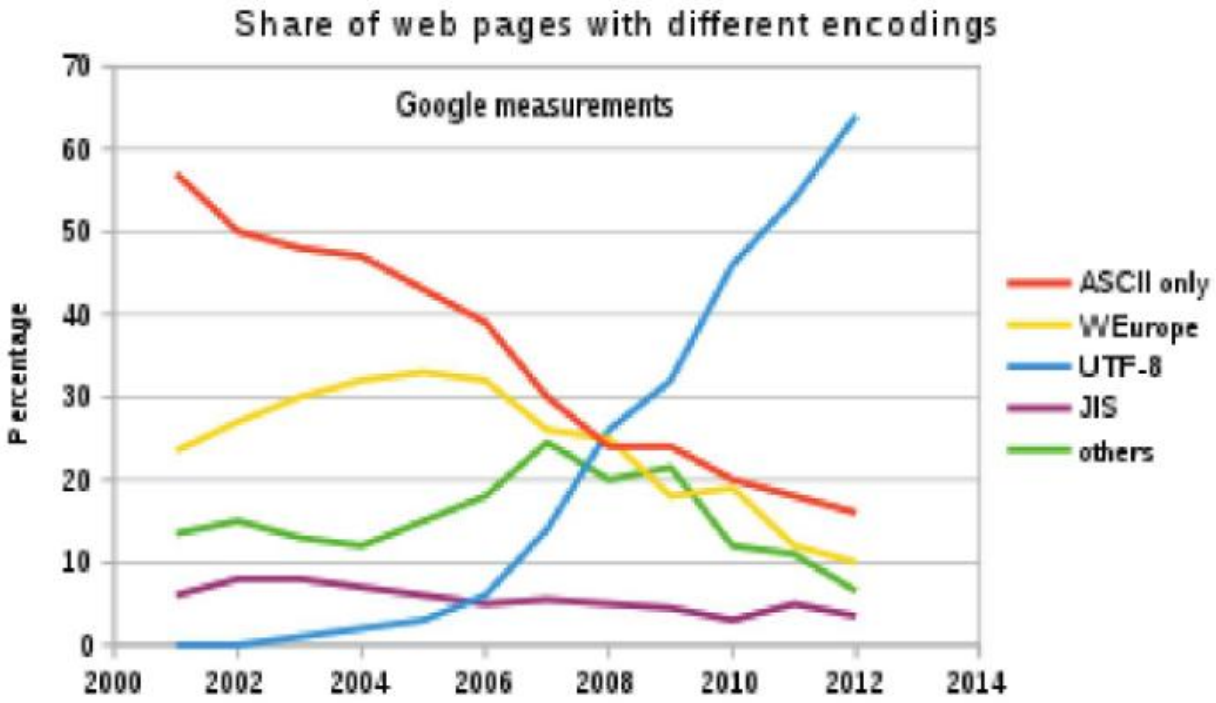
Codificación	Espacio por carácter	Para sistemas de escritura	Lenguas
ASCII	1 *byte* (7 bits)	Alfabeto latino	Inglés
ISO 8859-1 ISO 8859-15 Windows-1252	1 *byte*	Alfabeto latino	(Entre otras): inglés, español, francés, portugués, alemán, italiano, catalán, euskera, danés, gallego, holandés, noruego, sueco
UTF-8	De 1 a 4 *bytes*	Cualquiera	Cualquiera

La codificación más sencilla y antigua es **ASCII**. Utiliza un byte para representar cada carácter. Realmente solo utiliza siete bits, lo que permite representar 128 caracteres, más que suficiente para el idioma inglés. Adelantando un poco, se puede decir que **UTF-8** y todas las otras son compatibles con **ASCII**, lo que significa que cualquier texto codificado en **ASCII** se representa exactamente igual en las otras. En otras palabras, a menos que un texto contenga caracteres propios de algún idioma que no sea el inglés, tales como por ejemplo vocales acentuadas, se representará igual en **UTF-8** y en cualquier otra codificación.

Los problemas surgieron con los intentos descoordinados de dar soporte a otras lenguas. En lugar de un estándar para todas las lenguas, surgió un conjunto de estándares, cada uno dando soporte a un conjunto de lenguas. Así, el estándar **ISO 8859** incluye **ISO 8859-1**, que da soporte al español y a otras lenguas, pero también **ISO 8859-2**, **ISO 8859-3**, etc., cada uno de los cuales da soporte a un conjunto de lenguas. **ISO 8859-1** fue posteriormente revisado para dar lugar a **ISO 8859-15**, con soporte para el símbolo del euro (€) y algunas correcciones y mejoras. Pero **Microsoft** empezó a utilizar sus propias codificaciones no estándares. Entre ellas, está **Windows 1252**, que da soporte a las mismas lenguas que **ISO 8859-1** y, de hecho, es casi idéntica, pero no igual. Como curiosidad, el estándar **HTML5** incluye soporte para **Windows 1252**, considerado una variante de **ISO-8859-1**.

Unicode surgió como solución universal y definitiva para representar cualquier texto en cualquier idioma habido y por haber, y con cualquier sistema de escritura habido y por haber. **UTF-8** es una codificación de **Unicode** (hay otras, por ejemplo, **UTF-16**). Tiene la gran ventaja de que es compatible con **ASCII**, lo que no es el caso para **UTF-16**. Pero **UTF-8** no es compatible con **ISO 8859-1**. Esto significa que cualquier programa que asuma codificación **UTF-8** no mostrará correctamente textos codificados en **ISO 8859-1**, y en particular caracteres que no existen en inglés, tales como las vocales acentuadas o la letra ñ.

UTF-8 ha sido completamente adoptada en **Linux** como codificación estándar por defecto. No ha sido así en **Windows**. En **Windows**, los ficheros que contienen caracteres no representables mediante la codificación **Windows 1252** y similares se han venido codificando en una variante no estándar de **UTF-16**. Pero parece que **Windows** adoptará en breve **UTF-8** como codificación por defecto. En una actualización de **Windows**, del 10 de abril de 2018, apareció una opción *beta* para configurar la codificación de texto como **UTF-8**.



Distintas codificaciones para el texto de las páginas web. WEurope es ISO 8859-1

2.4. La clase File de Java

La versión de Java utilizada para este libro es Java SE 8, una versión LTS (**long term support**), es decir, con **soporte a largo plazo**. En los servidores web de Oracle se puede consultar la documentación de la biblioteca estándar de clases de Java SE 8.

Las clases que permiten trabajar con ficheros están en el paquete **java.io**.

*Se recomienda consultar en **http://docs.oracle.com/javase/8/docs/api/** la documentación de Java SE8 para más información acerca de cualquier que vayamos a utilizar. Arriba, a la izquierda, aparece la lista de paquetes por orden alfabético. En ella se puede localizar el paquete **java.io**. Si se pulsa sobre él, aparece debajo la lista de interfaces y clases que contiene el paquete.*

La clase **File** permite obtener información relativa a directorios y ficheros dentro de un sistema de ficheros y realizar diversas operaciones con ellos tales como borrar, renombrar, etc. En el siguiente cuadro se proporciona un resumen de la funcionalidad de esta clase, mostrando solo los principales métodos agrupados por categorías.

Métodos de la clase `File`

Categoría	Modificador/tipo	Método(s)	Funcionalidad
Constructor		`File(String ruta)`	Crea objeto `File` para la ruta indicada, que puede corresponder a un directorio o a un fichero.

Consulta de propiedades	`boolean`	`canRead()`
		`canWrite()`
		`canExecute()`
			Comprueban si el programa tiene diversos tipos de permisos sobre el fichero o directorio, tales como de lectura, escritura y ejecución (si se trata de un fichero). Para un directorio, `canExecute()` significa que se puede establecer como directorio actual.

	`boolean`	`exists()`	Comprueba si el fichero o directorio existe.
	`boolean`	`isDirectory()` `isFile()`	Comprueban si se trata de un directorio o de un fichero.
	`long`	`length()`	Devuelve longitud del fichero.
	`File`	`getParent()` `getParentFile()`	Devuelven el directorio padre.
	`String`	`getName()`	Devuelve nombre del fichero.
Enumeración	`String[]`	`list()`	Devuelve un *array* con los nombres de los directorios y ficheros dentro del directorio.
	`File[]`	`listFiles()`	Devuelve un *array* con los directorios y ficheros dentro del directorio.
Creación, borrado y renombrado	`boolean`	`createNewFile()`	Crea nuevo fichero.
	`static File`	`createTempFile()`	Crea nuevo fichero temporal y devuelve objeto de tipo `File` asociado, para poder trabajar con él.
	`boolean`	`delete()`	Borra fichero o directorio.
	`boolean`	`renameTo()`	Renombra fichero o directorio.
	`boolean`	`mkdir()`	Crea un directorio.
Otras	`java.nio.file.Path`	`toPath()`	Devuelve un objeto que permite acceder a información y funcionalidad adicional proporcionada por el paquete `java.nio`.

2.5. Gestión de excepciones en Java

Antes de seguir avanzando con el contenido, se incluye un breve repaso a la gestión de excepciones en el lenguaje Java. Cualquier programa escrito en Java debe realizar una adecuada gestión de excepciones. En este apartado se explicará lo más importante de la gestión de excepciones en Java, o al menos todo lo que se vaya a necesitar para el módulo.

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe su curso normal de ejecución. Una excepción podría producirse, por ejemplo, al dividir por cero.

2.5.1. Captura y gestión de excepciones

Cuando tiene lugar una excepción no gestionada, como al dividir por cero, aparte de mostrarse un mensaje de error, se aborta la ejecución del programa. Por ese motivo, el programa no mostraría el resultado de la ejecución. Cualquier fragmento de programa que pueda generar una excepción debería capturarlas y gestionarlas.

La salida del programa "dividir por cero" indica el tipo de excepción que se ha producido, a saber, **ArithmeticException**. Se puede capturar esta excepción con un sencillo bloque **try {} catch {}** y gestionarla.

*Es recomendable escribir los mensajes de error en la salida de error **System.err** en lugar de en la salida estándar **System.out**.*

Las excepciones son objetos de Java, pertenecientes a la clase **Exception** o a una subclase de ella. Esta clase tiene varios métodos interesantes para obtener información acerca de la excepción que se ha producido.

Métodos de la clase `Exception`

Modificador/tipo	Método(s)	Funcionalidad
`void`	`printStackTrace()`	Muestra información técnica muy detallada acerca de la excepción y el contexto en que se produjo. Lo hace en la salida de error, `System.err`. Al principio del desarrollo de un programa, y para programas de prueba, puede ser una buena opción utilizar esta función para mostrar información de todas las excepciones, y perfilar más adelante cómo se gestionan excepciones de tipos particulares.
`String`	`getMessage()`	Proporciona un mensaje detallado acerca de la excepción.
`String`	`getLocalizedMessage()`	Proporciona una descripción localizada (es decir, traducida a la lengua local) de la excepción.

2.5.2. Gestión diferenciada de distintos tipos de excepciones

En un bloque try {} catch {} se pueden gestionar por separado distintos tipos de excepciones. Es conveniente incluir un manejador para Exception al final para que ninguna excepción se quede sin gestionar.

PARA SABER MÁS

*En aplicaciones profesionales, la práctica habitual es utilizar herramientas de logging (de registro), y no solo para mensajes de error. Los mensajes se suelen registrar en ficheros. Estas herramientas permiten generar de forma diferenciada distintos tipos de mensaje, típicamente, y por orden de importancia, de mayor a menor: **error**, **warning** (de aviso), **info** (informativo) y **debug** (para depuración). Permiten configurar el nivel de logging, de manera que solo se registren los mensajes a partir del nivel de importancia establecido. Si este se establece como **warning**, se mostrarían los de tipo **warning** y **error**. Entre las herramientas más ampliamente utilizadas para Java están:*

- *Java Logging API (http://docs.oracle.com/javase/8/docs/technotes/guides/logging)*
- *Log4j (http://logging.apache.org/log4j)*

2.5.3. Declaración de excepciones lanzadas por un método de clase

Si el compilador es capaz de determinar que un método de una clase puede originar un tipo de excepción, pero no lo gestiona mediante un bloque catch(){}, la compilación terminará con un error. Una posibilidad entonces es gestionar la excepción en el método mediante un bloque catch(){}. La otra es añadir el modificador throws seguido de la clase de excepción.

2.5.4. Excepciones, inicialización y liberación de recursos: bloques finally y try con recursos

Es muy frecuente que un bloque de programa de Java esté estructurado de la siguiente forma:

Inicialización y asignación de recursos

Cuerpo

Finalización y liberación de recursos

Este bloque puede estar dentro de un método de clase, como por ejemplo, el método main(), o de un bucle.

La primera y última parte deben ejecutarse siempre, independientemente de los errores que puedan suceder durante la ejecución del cuerpo. El bloque anterior, con gestión de excepciones, podría quedar de la siguiente manera:

Inicialización y asignación de recursos

try {

Cuerpo

} catch (Excepcion_Tipo_1 e1) {

Gestión de excepción de tipo 1

} catch (Excepcion_Tipo_2 e2) {

Gestión de excepción de tipo 2

} catch (Exception e) {

Gestión del resto de tipos de excepciones

}

Finalización y liberación de recursos

Pero es muy frecuente que, cuando sucede algún error en mitad del cuerpo, se quiera terminar inmediatamente la ejecución, bien sea con una sentencia **return** (en un método de clase), o con **break** o **continue** (dentro de un bucle, para salir de él o para saltar a la siguiente iteración). De esta manera no se ejecuta la parte final para finalización y liberación de recursos. Pero si se pone esta parte dentro de un bloque **finally** {}, se ejecutará justo antes de abandonar.

Inicialización y asignación de recursos

try {

Cuerpo

```
} catch(Excepcion_Tipo_1 e1) {
Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
Gestión de excepción de tipo 2
} catch(Exception e) {
Gestión del resto de tipos de excepciones
}
}
finally {
Finalización y liberación de recursos
}
```

Los bloques **try** con recursos son de utilidad para simplificar la gestión de recursos de clases que implementan una de las interfaces **Closeable** o **AutoCloseable**. Para estos se invocará automáticamente el método **close()** en el bloque **finally**. Si no hay un bloque **finally**, se puede entender que existe uno vacío.

```
Inicialización y asignación de recursos
try (T1 r1=new T1(); T2 r2=new T2()) {
// T1, T2 implementan Closeable o AutoCloseable
Cuerpo
} catch(Excepcion_Tipo_1 e1) {
Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
Gestión de excepción de tipo 2
} catch(Exception e) {
Gestión del resto de tipos de excepciones
}
}
finally {
Finalización y liberación de recursos
// No es necesario r1.close()
// No es necesario r2.close()
}
```

2.6. Formas de acceso a los ficheros

Existen dos formas de acceder a los contenidos de los ficheros. A saber:

1. **Acceso secuencial.** Se accede comenzando desde el principio del fichero. Para llegar a cualquier parte del fichero, hay que pasar antes por todos los contenidos anteriores, empezando desde el principio del fichero.
2. **Acceso aleatorio.** Se accede directamente a los datos situados en cualquier posición del fichero.

Ambas formas de acceso se pueden utilizar para operaciones tanto de lectura como de escritura. La mayoría de los medios de almacenamiento permiten el acceso secuencial y el acceso aleatorio a los ficheros que almacenan. Pero en algunos casos podría ser posible solo el acceso secuencial. Este sería el caso si el fichero está almacenado en una cinta magnética, si bien este medio de almacenamiento está hoy en desuso. Si se considera el concepto más general de flujo de datos o **stream**, del que un fichero sería un caso particular, existen más casos en los que solo es posible el acceso secuencial. Cuando dos aplicaciones se comunican mediante una conexión de red, por ejemplo, la aplicación de origen envía los datos secuencialmente, y la de destino los recibe secuencialmente. En muchos lenguajes de programación, entre ellos Java, se pueden utilizar las mismas funciones para enviar y recibir datos por una conexión de red que para escribir y leer datos en ficheros.

A continuación, el texto de la imagen:

2.7. Operaciones sobre ficheros con Java

Independientemente del tipo de fichero (binario o de texto) y del tipo de acceso (secuencial o aleatorio), las operaciones básicas sobre ficheros son en lo esencial iguales, y se explicarán en este apartado. En los apartados siguientes se introducirán las particularidades de las clases que proporciona Java para diversos tipos de operaciones con diversos tipos de ficheros, y las operaciones adicionales que cada una permite realizar.

El mecanismo de acceso a un fichero está basado en un puntero y en una zona de memoria que se suele llamar **buffer**. El puntero siempre apunta a un lugar del fichero, o bien a una posición especial de fin de fichero, que a veces se denomina **EOF** (del inglés **end of file**). Esta se puede entender que está situada inmediatamente a continuación del último **byte** del fichero. Todas las clases para operar con ficheros disponen de las siguientes operaciones básicas:

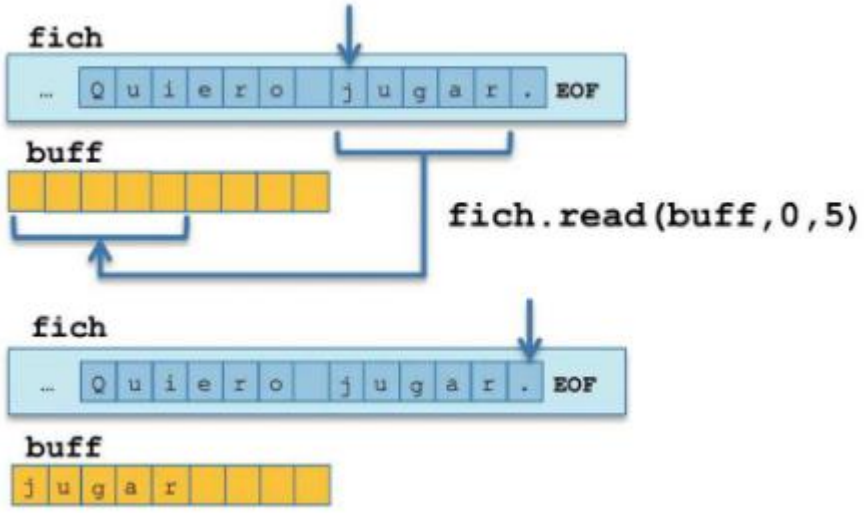
1. **Apertura**. Antes de hacer nada con un fichero, hay que abrirlo. Esto se hace al crear una instancia de una clase que se utilizará para operar con él.
2. **Lectura**. Mediante el método **read()**. Consiste en leer contenidos del fichero para volcarlos a memoria y poder trabajar con ellos. El puntero se sitúa justo después del último carácter leído.
3. **Salto**. Mediante el método **skip()**. Consiste en hacer avanzar el puntero un número determinado de **bytes** o caracteres hacia delante.
4. **Escritura**. Mediante el método **write()**. Consiste en escribir contenidos de memoria en un lugar determinado del fichero. El puntero se sitúa justo después del último carácter escrito.
5. **Cierre**. Mediante el método **close()**. Para terminar, hay que cerrar el fichero.

La diferencia entre el acceso secuencial y el acceso aleatorio es que, con el último, se puede, en cualquier momento, situar el puntero en cualquier lugar del fichero. En cambio, con el acceso secuencial solo se mueve el cursor tras realizar operaciones de lectura, escritura o salto.

2.7.1. Operaciones de lectura

Cuando se lee desde el fichero, hay que indicar el **buffer**, que recibirá los datos que se lean desde él. Si no se indica el número de **bytes** o de caracteres para leer, se leerá hasta llenar el **buffer**. Lógicamente, no se leerá nada si el puntero apunta a la posición **EOF** (final del fichero).

Los ejemplos que siguen, por claridad, son para un fichero de texto en el que cada carácter ocupa un **byte**. Pero la operación de lectura puede realizarse para un fichero tanto binario como de texto. En el primer caso, se leerán **bytes**, y en el segundo, caracteres, cada uno de los cuales puede estar formado por un número variable de **bytes**, dependiendo del carácter y de la codificación empleada para el texto.



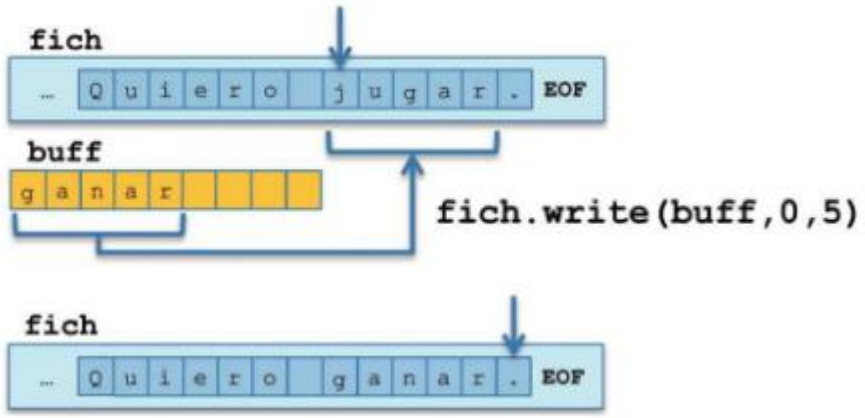
Lectura desde fichero

2.7.2. Operaciones de escritura

Cuando se escribe en el fichero, hay que indicar el **buffer** y el número de **bytes** para leer. Desde el **buffer** se transfieren los datos a la posición a la que apunta el puntero.

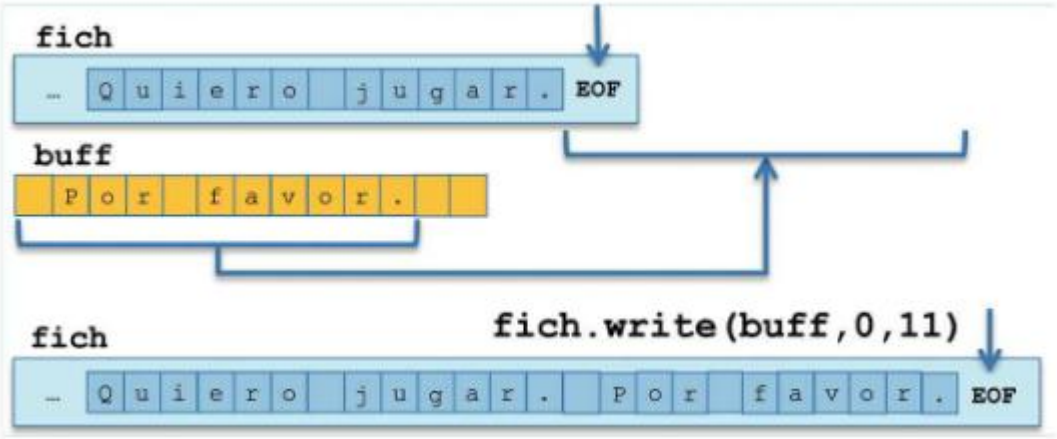
Si el puntero apunta al final del fichero, se añaden los datos al final de este.

Para concluir con las operaciones básicas, poco más se puede hacer que lo visto hasta ahora. No existe, por ejemplo, ninguna forma directa de insertar datos en medio de un fichero, de eliminar un fragmento de un fichero, ni de sustituir los contenidos de un fragmento de un fichero por otros, como no sea que tengan la misma longitud.



Escritura de un fichero cuando el puntero apunta en medio del fichero

Esto es así en Java y en prácticamente todos los lenguajes de programación, y se debe a que las implementaciones de los sistemas de ficheros más habituales no proporcionan ninguna manera directa para hacer operaciones de este tipo. Pero se puede hacer utilizando ficheros auxiliares, como se verá más adelante.



Escritura de un fichero cuando el puntero apunta al final del fichero

2.8. Acceso secuencial a ficheros en Java

Las operaciones con ficheros secuenciales en Java se realizan utilizando **flujos** (**streams**, en inglés) mediante clases del paquete **java.io**. Un flujo es una abstracción de alto nivel para cualquier secuencia de datos. Los ficheros secuenciales son flujos, lo mismo que los datos enviados a través de una conexión de red, o la información contenida en una secuencia de posiciones consecutivas de memoria, o la entrada y la salida estándar y de error de un programa en ejecución. Este paquete saca partido de la herencia, y proporciona clases con funcionalidad genérica de entrada y salida, común para cualquier tipo de flujo, y clases con funcionalidad específica para los distintos tipos de flujos.

Existe una jerarquía de clases derivada de cuatro clases, correspondientes a las cuatro combinaciones posibles, considerando, por una parte, flujos binarios y de texto y, por otra, flujos de entrada y de salida.

2.8.1. Clases relacionadas con flujos de datos

Para leer de un fichero o escribir en un fichero hay que crear un flujo (**stream**) asociado al mismo. El tipo de flujo para crear será distinto según se trate de un fichero binario o de texto.

Los flujos binarios no plantean mayor problema: se leen **bytes** de un flujo binario hacia una variable de tipo **byte[]**, o se escriben **bytes** de una variable de tipo **byte[]** a un flujo binario.

Cuando se lee o escribe en flujos de texto, en cambio, todo es más complicado, porque entran en juego las codificaciones de texto. Todo funcionará bien mientras se trabaje con ficheros de texto con la codificación que Java asume por defecto. Normalmente será así, pero puede no serlo.

La codificación que utiliza Java por defecto para flujos de texto la toma de la configuración de la máquina virtual de Java, o en su defecto de la configuración del sistema operativo, o asume UTF-8. En la práctica, normalmente será UTF-8, que es la más habitual hoy en día. Esta codificación utiliza de uno a cuatro **bytes** para representar un carácter. Uno para todo lo que esté en el antiguo código ASCII. Dos para todo lo que no existe en inglés, como vocales acentuadas, la letra **ñ**, etc. Y hasta cuatro con algunos sistemas de escritura no basados en el alfabeto latino.

En cambio, para almacenar texto en memoria, Java utiliza UTF-16. Por tanto, las lecturas y escrituras con ficheros de texto suelen llevar implícito un proceso de recodificación. UTF-16 utiliza dos o cuatro **bytes** para representar un carácter o, lo que es lo mismo, uno o dos **char** (que en Java son dos **bytes**). Una cadena de texto en Java se almacena, pues, en un **array char[]** o en un **String** codificado en UTF-16 como una secuencia de caracteres, cada uno representado por dos **bytes** (normalmente) o por cuatro (con algunos sistemas de escritura no basados en el alfabeto latino).

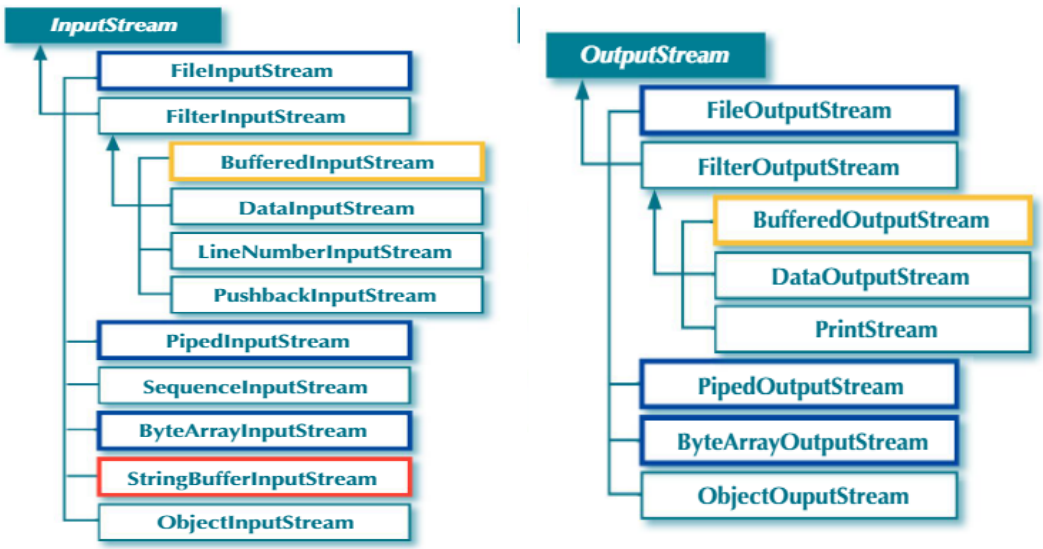
RECUERDA

*Java codifica el texto en memoria en UTF-16, en variables de tipo **String** o **char[]**. La mayoría de los ficheros de texto están codificados en UTF-8, y normalmente Java utiliza esta codificación por defecto para leer y escribir en ellos. Por lo tanto, las operaciones de lectura o escritura de texto con ficheros de texto llevan implícito, generalmente, un proceso de recodificación.*

Para lectura y escritura en flujos, Java proporciona dos jerarquías de clases: una para flujos binarios y otra para flujos de texto.

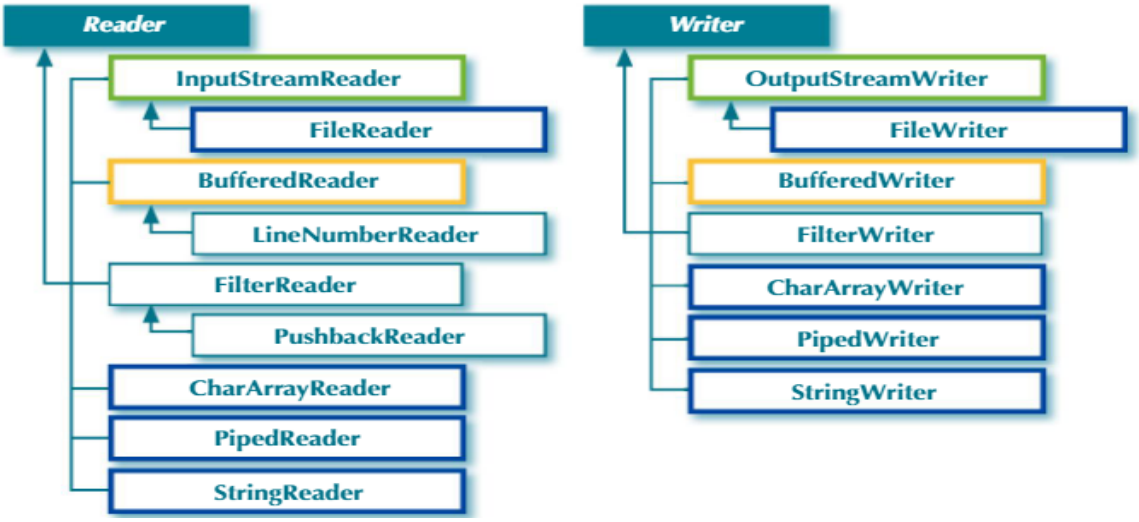
Clases para flujos binarios:

- InputStream y OutputStream



Clases para flujos de texto:

- Reader y Writer



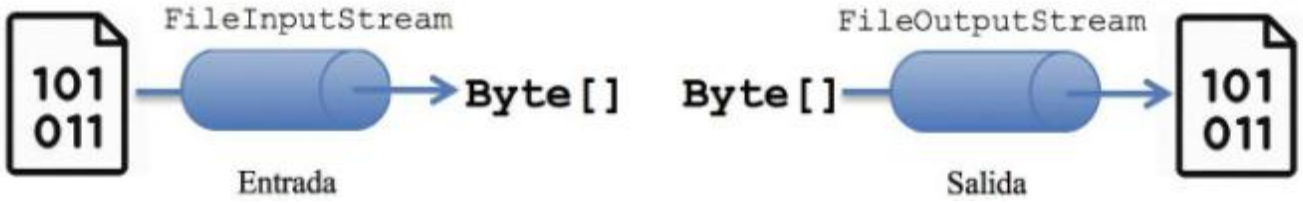
En estas dos jerarquías, algunas clases ofrecen la funcionalidad básica de lectura y escritura en flujos asociados a distintos tipos de fuentes de datos. Otras ofrecen recodificación de texto. Otras ofrecen **buffering**, que permite aumentar la velocidad de las operaciones de lectura y escritura, y hace posible leer y escribir líneas de texto para los ficheros de texto. Todas estas clases se pueden componer unas sobre otras, de manera que cada una añada funcionalidad nueva sobre la proporcionada por las anteriores.

- Las clases que ofrecen la funcionalidad básica de lectura y escritura son las representadas con borde azul marino. Java usa una nomenclatura sistemática para ellas. El nombre indica primero el tipo de fuente (**File** para fichero, **ByteArray** para memoria, **Piped** para tubería). Después, si es de entrada (**Input** o **Reader**) o de salida (**Output** o **Writer**). Si acaba con **Stream** es para flujos de **bytes**, y si no, para flujos de texto.
- Las clases que ofrecen recodificación de texto son las representadas con borde verde claro.
- Las clases que ofrecen **buffering** son las representadas con borde verde oscuro.

Clases básicas para entrada y salida de flujos.

	Fuente de datos	Lectura	Escritura
Flujo binario	Ficheros	FileInputStream	FileOutputStream
	Memoria (byte[])	ByteArrayInputStream	ByteArrayOutputStream
	Tuberías	PipedInputStream	PipedOutputStream
Flujo de texto	Ficheros	FileReader	FileWriter
	Memoria (char[])	CharArrayReader	CharArrayWriter
	Memoria (String)	StringReader	StringWriter
	Tuberías	PipedReader	PipedWriter

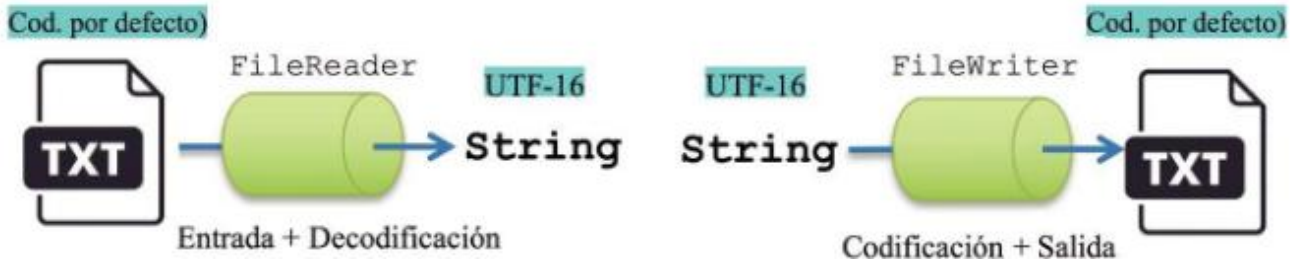
A continuación, se dan algunos ejemplos de cómo se crean objetos de estas clases. Para más información acerca de ellas, se puede consultar la documentación de Java.




```
FileInputStream fis = new FileInputStream("f.bin");
```

```
FileOutputStream fos = new FileOutputStream("f.bin");
```

Lectura y escritura en ficheros binarios




```
FileReader fr = new FileReader("f.txt");
```

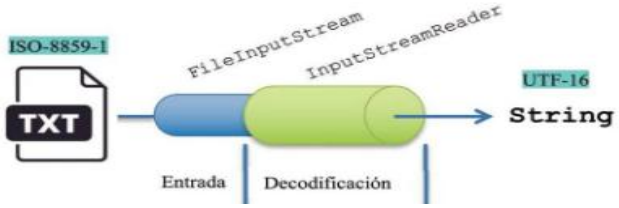
```
FileWriter fw2 = new FileWriter("f.txt");
```

Lectura y escritura en ficheros de texto con la codificación por defecto

A continuación, el texto de la imagen:

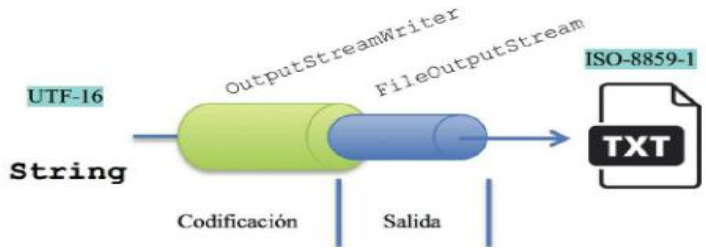
2.8.2. Clases para recodificación

Las clases **InputStreamReader** y **OutputStreamWriter** sirven de enlace entre ambas jerarquías: la de flujos binarios y la de flujos de texto. Dado que convierten flujos binarios en flujos de texto y viceversa, es necesario especificar una codificación a su constructor. Se pueden entender como clases que permiten recodificar texto, es decir, leer o escribir texto en ficheros con una codificación diferente a la codificación por defecto (que, como ya se ha comentado, suele ser UTF-8).




```
InputStreamReader fisr = new InputStreamReader(new FileInputStream("f_ISO-8859_1.txt"), "ISO-8859-1");
```

Lectura desde ficheros de texto con codificación distinta a la codificación por defecto




```
OutputStreamWriter fosw = new OutputStreamWriter(new FileOutputStream("f_ISO-8859_1.txt"), "ISO-8859-1");
```

Escritura a ficheros de texto con codificación distinta a la codificación por defecto

2.8.3. Clases para buffering

Existen algunas clases que proporcionan **buffering** sobre el servicio proporcionado por las clases anteriores. El **buffering** es una técnica que permite acelerar las operaciones de lectura o escritura utilizando una zona de intercambio en memoria llamada **buffer**. Estas clases permiten además leer líneas de texto y escribir líneas de texto en ficheros de texto.

TOMA NOTA

*En general conviene utilizar las clases que proporcionan **buffering**. El rendimiento aumenta para ficheros grandes y cuando se realizan lecturas o escrituras consecutivas en posiciones contiguas, que es lo habitual. Cuando no es así, la merma en el rendimiento no es significativa.*

*El **buffer** siempre representa el contenido actual de una sección del fichero. Cuando se lee de un fichero, la lectura no se limita a los datos solicitados, sino que se traen datos suficientes para llenar el **buffer**. Si la siguiente lectura del fichero es en posiciones consecutivas, lo que es muy habitual, los datos ya estarán en el **buffer**, y así se ahorra un nuevo acceso al fichero. Se puede utilizar un **buffer** de manera análoga para las operaciones de escritura. En lugar de escribir directamente en el fichero, se escribe en el **buffer**. Solo se vuelcan los contenidos del **buffer** en el fichero cuando una operación de escritura afecta a una parte del fichero fuera de la sección representada en el **buffer**, o cuando otro programa lee información de esa sección. A la actualización del fichero con los contenidos del **buffer** se le llama en inglés **flushing**.*

El buffer siempre representa el contenido actual de una sección del fichero. Cuando se lee de un fichero, la lectura no se limita a los datos solicitados, sino que se traen datos suficientes para llenar el buffer. Si la siguiente lectura del fichero es en posiciones consecutivas, lo que es muy habitual, los datos ya estarán en el buffer, y así se ahorra un nuevo acceso al fichero. Se puede utilizar un buffer de manera análoga para las operaciones de escritura. En lugar de escribir directamente en el fichero, se escribe en el buffer. Solo se vuelcan los contenidos del buffer en el fichero cuando una operación de escritura afecta a una parte del fichero fuera de la sección representada en el buffer, o cuando otro programa lee información de esa sección. A la actualización del fichero con los contenidos del buffer se le llama en inglés flushing.

Clases que proporcionan buffering para entrada y salida a flujos

	Lectura	Escritura
Flujo binario	BufferedInputStream	BufferedOutputStream
Flujo de texto	BufferedReader	BufferedWriter

Cada una de las clases anteriores proporciona buffering para objetos de una clase cuyo nombre viene después de **Buffered**, y además es subclase de ella, por lo que se puede usar en su lugar. Se podrían cambiar algunos de los ejemplos anteriores para utilizar buffering:

Flujo sin buffering y con buffering

Flujo sin buffering	**Flujo con buffering**
new FileInputStream("f.bin")	new BufferedInputStream(new FileInputStream("f.bin"))
new FileOutputStream("f.bin")	new BufferedOutputStream(new FileOutputStream("f.bin"))
new FileReader("f.txt")	new BufferedReader(new FileReader("f.txt"))
new FileWriter("f.txt")	new BufferedWriter(new FileWriter("f.txt"))

Como ya se ha dicho, las clases que proporcionan buffering para ficheros de texto permiten además leer y escribir líneas de texto.

Métodos para lectura y escritura de líneas en clases para buffering con ficheros de texto

Clase	**Método**	**Funcionalidad**
BufferedReader	String readLine()	Lee hasta el final de la línea actual.
BufferedWriter	void newLine()	Escribe un separador de líneas. El separador de líneas puede depender del sistema operativo, y suele ser distinto en Linux y en Windows. El método readLine() de BufferedReader tiene en cuenta estas particularidades.

2.8.4. Operaciones de lectura para flujos de entrada

Son similares para todas las clases que implementan flujos (streams) de entrada en Java. Las funciones read de clases que heredan de InputStream leen bytes, mientras que las que heredan de Reader leen caracteres. Según la variante, leen un byte o un carácter hasta llenar el buffer o el número de bytes o caracteres indicados, que se copiarán en la posición indicada (offset) dentro del buffer. Devuelven el número de bytes o caracteres leídos, o -1 si no se pudo leer nada porque el puntero estaba al final del fichero. Las funciones skip saltan el número indicado de bytes o caracteres, aunque podrían ser menos si se alcanza el final del fichero. En cualquier caso, devuelven el número de bytes o de caracteres que se han saltado.

Con ficheros de texto, para leer líneas se puede utilizar el método readLine() de un BufferedReader construido sobre un FileReader.

Métodos para lectura de las clases para gestión de flujos de entrada

InputStream	**Reader**
int read()	int read()
int read(byte[] buffer)	int read(char[] buffer)
int read(byte[] buffer, int offset, int longitud)	int read(char[] buffer, int offset, int longitud)
	int read(CharBuffer buffer)
long skip(long n)	long skip(long n)
	BufferedReader
	String readLine()

2.8.5. Operaciones de escritura para flujos de salida

Son similares para todas las clases que implementan flujos (**streams**) de salida en Java. Las funciones **write** de clases que heredan de **OutputStream** escriben **bytes**, mientras que las que heredan de **Writer** escriben caracteres. Según la variante, escriben un **byte** o un carácter, o todos los contenidos del **buffer**, o el número de **bytes** o caracteres indicados a partir de la posición indicada. Si se alcanza el fin del fichero, siguen escribiendo. Las que heredan de **Writer** tienen métodos para escribir los contenidos de un **String**, y métodos **append** para añadir al final del fichero.

Una característica muy útil de las clases **FileOutputStream** y **Writer** es que disponen de constructores con un parámetro que permite abrir ficheros para añadir contenidos al final (**append** en inglés). También disponen de constructores sin ese parámetro.

FileOutputStream(File file, boolean append)

FileOutputStream(String nombreFichero, boolean append)

Writer (File file, boolean append)

Writer(String nombreFichero, boolean append)

Métodos para escritura de clases para gestión de flujos de salida

OutputStream	**Writer**
void write(int b)	**void write(int c)**
void write(byte[] buffer)	**void write(char[] buffer)**
void write(byte[] buffer, int offset, int longitud)	**void write(char[] buffer, int offset, int longitud)**
	void write(String str)

OutputStream	Writer
	void write(String str, int offset, int longitud)
	Writer append(char c)
	Writer append(CharSequence csq)
	Writer append(CharSequence csq, int offset, int longitud)
	BufferedWriter
	void newLine()

No es posible eliminar o reemplazar contenidos de un fichero directamente utilizando solo flujos, porque para ello es necesario leer y escribir en el mismo fichero. Se puede hacer utilizando **ficheros auxiliares**, que se pueden crear con createTempFile() de la clase File.

Los ficheros temporales creados con **createTempFile()** normalmente se crean en un directorio especial para ficheros temporales del sistema operativo (**/tmp** en Linux). Es conveniente borrarlos al final si no se necesitan más (no es el caso en el ejemplo anterior, porque se renombra para pasar a ser un fichero definitivo). Si no se hace, debería ser el propio sistema operativo el que los elimine más adelante, por ejemplo, la próxima vez que se reinicie el sistema o pasado un tiempo. Pero eso depende del sistema operativo y de cómo esté configurado. Algunos ordenadores, como por ejemplo servidores, no se reinician durante periodos muy largos de tiempo.

2.9. Operaciones con ficheros de acceso aleatorio en Java

Para el acceso aleatorio a ficheros se utiliza la clase **RandomAccessFile**.

Las novedades fundamentales que aporta respecto a lo ya visto hasta ahora son:

1. En todo momento el cursor se puede situar en cualquier posición mediante la función **seek()**. En eso consiste el acceso aleatorio.
2. Sobre el fichero se pueden realizar operaciones tanto de lectura como de escritura.

A pesar de estas nuevas posibilidades, las operaciones con ficheros siguen teniendo muchas limitaciones que no vienen del lenguaje Java ni del paquete **java.io**, sino de los sistemas de ficheros habituales. A saber, no es posible eliminar o insertar bloques de **bytes** o de caracteres en mitad de un fichero, ni reemplazar un bloque de un fichero por otro, a no ser que tengan exactamente el mismo tamaño en **bytes**. Para ello habrá que utilizar ficheros temporales auxiliares, de manera similar a como se ha visto en el ejemplo anterior.

En la siguiente tabla se proporciona un resumen de los principales métodos de esta clase. El resto de los métodos se puede consultar en la documentación del paquete **java.io**.

Principales métodos de la clase RandomAccessFile

Método	Funcionalidad
RandomAccessFile(File file, String mode) **RandomAccessFile(String name, String mode)**	Constructor. Abre el fichero en el modo indicado, si se dispone de permisos suficientes. **r**: modo de solo lectura. **rw**: modo de lectura y escritura. **rwd, rws**: Como **rw** pero con escritura síncrona. Esto significa que todas las operaciones de escritura (de datos con **rwd** y de datos y metadatos con **rws**) deben haberse completado cuando termina la función. Esto puede hacer que la llamada a la función tarde más, pero asegura que no se pierde información crítica ante una caída del sistema.
void close()	Cierra el fichero. Es conveniente hacerlo siempre al final.
void seek(long pos)	Posiciona el puntero en la posición indicada.
int skipBytes(int n)	Intenta avanzar el puntero el número de **bytes** indicado. Se devuelve el número de **bytes** que se ha avanzado. Podría ser menor que el solicitado, si se alcanza el fin del fichero.
int read() **int read(byte[] buffer)** **int read(byte[] buffer, int offset, int longitud)**	Lee del fichero. Según la variante, un **byte** o hasta llenar el **buffer**, o el número de **bytes** indicados, que se copiarán en la posición indicada (**offset**) del **buffer**. Devuelve el número de **bytes** leídos, **-1** si no se pudo leer nada porque el puntero estaba al final del fichero.
Void readFully(byte[] buffer) **readFully(byte[] buffer, int offset, int longitud)**	Como **int read(byte[] b)**, pero si no se puede leer hasta llenar el **buffer**, o el número de **bytes** indicado, porque se llega al final del fichero, se lanza la excepción **IOException**. Útil cuando se sabe que se podrá leer hasta llenar el **buffer** o el número de **bytes** indicados. En cualquier caso, la eventualidad de que no se pueda completar la lectura se puede gestionar capturando la excepción.
String readLine()	Lee hasta el final de la línea de texto actual.
void write(int b) **void write(byte[] buffer)** **void write(byte[] buffer, int offset, int longitud)**	Escribe en el fichero. Según la variante, un **byte**, o todos los contenidos del **buffer**, o el número de **bytes** indicados a partir de la posición indicada

Método	Funcionalidad
	(**offset**). Si alcanza el fin del fichero, siguen escribiendo.

2.10. Organizaciones de ficheros

Una organización de ficheros es una manera de organizar y estructurar los datos dentro de los ficheros, de manera que se puedan interpretar correctamente sus contenidos. El último programa de ejemplo para acceso aleatorio a ficheros almacena los da[1]tos de clientes en registros de longitud fija, compuestos por campos de longitud fija. Esta es una organización muy habitual. También es muy habitual que exista un campo, o un conjunto de campos, que identifique cada registro, de manera que no pueda existir más de un registro con los mismos valores para la clave. Se denomina **clave** a este campo o conjunto de campos. En el ejemplo anterior, en el que cada registro almacena los datos de un cliente, la clave sería el DNI del cliente.

Algunas organizaciones de ficheros pueden incluir estructuras complementarias en el propio fichero o en ficheros auxiliares. Por ejemplo, se podría incluir en el propio fichero un bloque de cabecera antes de la secuencia de campos o un bloque de cierre después. En dichos bloques se podría incluir el número de registros que contiene, la longitud de cada registro, la definición de los campos, etc. Algunas organizaciones de ficheros pueden incluir estructuras complementarias en ficheros auxiliares, como, por ejemplo, índices para acelerar las operaciones de consulta.

En definitiva, hay muchísimas posibles organizaciones de los contenidos de un fichero. Pero este apartado se centrará en dos muy sencillas, útiles y representativas: la organización secuencial y la organización secuencial indexada, basadas ambas en registros de longitud fija.

Con todo lo que se ha aprendido en este capítulo, con los ejemplos proporcionados y con la documentación de Java SE 8, se está en perfectas condiciones para desarrollar clases que implementen estas organizaciones de ficheros.

2.10.1. Organización secuencial

Los registros que forman el fichero se almacenan uno tras otro y no están ordenados de ninguna manera. No hay ningún mecanismo para localizar directamente un registro dado o para agilizar las búsquedas. La única manera es leer los registros uno a uno hasta encontrar el que se está buscando o hasta llegar al final del fichero. La figura 1.4 muestra los contenidos de un fichero con datos de clientes y con registros de longitud fija, siendo la clave el DNI.

La principal ventaja que tiene esta organización es su sencillez. Según el uso que se le vaya a dar al fichero, esto podría compensar sus inconvenientes, a saber:

- Las búsquedas son muy ineficientes. Para buscar cualquier registro, es necesario recorrer secuencialmente el fichero desde el principio hasta que se encuentre el registro que se busca o hasta llegar al final del fichero sin encontrarlo. Ordenar los registros evitaría tener que recorrer todos los ficheros cuando no está el registro que se busca. Pero ordenar por DNI no servirá de nada si se busca por nombre, por ejemplo. Además, mantener el fichero siempre ordenado complica las operaciones de inserción, borrado o modificación de registros (esto último en el caso en que se modifique el campo por el que está ordenado el fichero).

- El borrado de un registro es muy ineficiente porque obliga a correr una posición hacia atrás todos los registros siguientes. Una posible mejora podría ser marcar el registro como borrado mediante alguna marca especial. Por ejemplo, un carácter determinado en la primera posición, o dejar en blanco los campos que forman la clave, en este caso el DNI.
- La inserción de registros sería muy eficiente. Bastaría añadir el nuevo registro al final. A no ser, como ya se ha visto, que el fichero estuviera ordenado.

2.10.2. Organización secuencial indexada

Esta organización permite búsquedas muy eficientes por cualquier campo que se quiera, a cambio de crear y mantener un fichero de índice para ese campo.

Un fichero de índice no es más que un fichero secuencial ordenado cuyos registros contienen dos campos: uno para un valor del campo para el que se crea el índice, y otro que indica la posición en el fichero secuencial.

Se pueden crear todos los índices que se quiera. Podría crearse uno por DNI, otro por nombre, o por el campo que se quiera. Podrían crearse también índices compuestos por más de un campo.

Utilizando esta organización, no es necesario reorganizar el fichero principal cuando se añaden nuevos registros o se modifican los que ya hay, solo es necesario reorganizar los índices, que son ficheros mucho más pequeños que el fichero principal. El marcar los registros como borrados en lugar de eliminarlos puede hacer innecesario reorganizar los índices cuando se elimina un registro.

El beneficio para las consultas que suponen los índices se consigue a cambio de espacio de almacenamiento y de tener que reorganizar todos los índices cuando se realizan operaciones de inserción, borrado o modificación. Hay que tenerlo en cuenta para evitar crear índices si los beneficios no compensan los inconvenientes.

Resumen

- Un fichero consiste en una secuencia de **bytes**. Un fichero se identifica por su nombre y el directorio donde está situado dentro de la jerarquía de directorios de un sistema de ficheros.
- Todos los sistemas para persistencia de datos almacenan los datos, en última instancia, en ficheros.
- Los sistemas para persistencia de datos basados en ficheros fueron relegados desde la década de los ochenta por los sistemas de bases de datos relacionales.
- Los ficheros pueden ser de texto o binarios. Los ficheros de texto contienen solo texto representado mediante una secuencia de **bytes**. Una codificación de texto es un método que permite representar un texto como una secuencia de **bytes**. La codificación de texto más ampliamente utilizada, y cada vez más, es UTF-8.
- El lenguaje de programación Java proporciona un completo soporte para operaciones con ficheros y directorios en el paquete **java.io**. Cuando se utiliza Java para operaciones con ficheros, y en general para cualquier cosa, hay que gestionar apropiadamente las excepciones que puedan producirse.
- La clase **File** permite acceder a propiedades de directorios y ficheros, así como crearlos, borrarlos, copiarlos y cambiar su ubicación dentro de la jerarquía de directorios de un sistema de ficheros.
- Hay dos formas fundamentales de acceso a ficheros: **acceso secuencial** y **acceso aleatorio**. Con acceso secuencial, para leer información en cualquier posición dentro del

fichero, es necesario leer antes la información en todas las posiciones anteriores. Con acceso aleatorio es posible leer directamente la información presente en cualquier posición del fichero.

- Para el **acceso secuencial** en Java se utilizan **streams**. Un fichero es un tipo particular de **stream**. Java proporciona cuatro jerarquías de clases para **streams**. Corresponden a las cuatro combinaciones de valores posibles entre **streams**, por una parte, de entrada y de salida y, por otra, binarios y de texto. Para **streams** binarios las clases de origen de las jerarquías son **InputStream** y **OutputStream**, y para **streams** de texto, **Reader** y **Writer**. Las clases **BufferedInputStream** y **BufferedOutputStream** proporcionan **buffering** para **streams** binarios, lo que permite acelerar las operaciones de entrada y salida, respectivamente. Las clases **BufferedReader** y **BufferedWriter** hacen lo propio para **streams** de texto, y permiten, además, leer y escribir, respectivamente, líneas de texto. Las clases **InputStreamReader** y **OutputStreamWriter** proporcionan recodificación de texto y sirven de enlace entre las jerarquías para **streams** binarios y de texto.
- Para el **acceso aleatorio** a ficheros, Java proporciona la clase **RandomAccessFile**. Un **RandomAccessFile** solo proporciona operaciones para lectura y escritura de **bytes**, lo que no significa que no se puedan utilizar para el acceso aleatorio a ficheros de texto.
- Las organizaciones más sencillas para almacenar datos en ficheros están basadas en registros de longitud fija, en los que cada registro está a su vez formado por varios campos de longitud fija. En cada registro puede existir un campo clave, de manera que no pueden existir dos registros en el fichero con el mismo valor para el campo clave. La organización más sencilla es la **organización secuencial**. Para acelerar las búsquedas se pueden utilizar ficheros de **índice externos**, que permiten acceder a los registros contenidos en el fichero en un orden determinado.