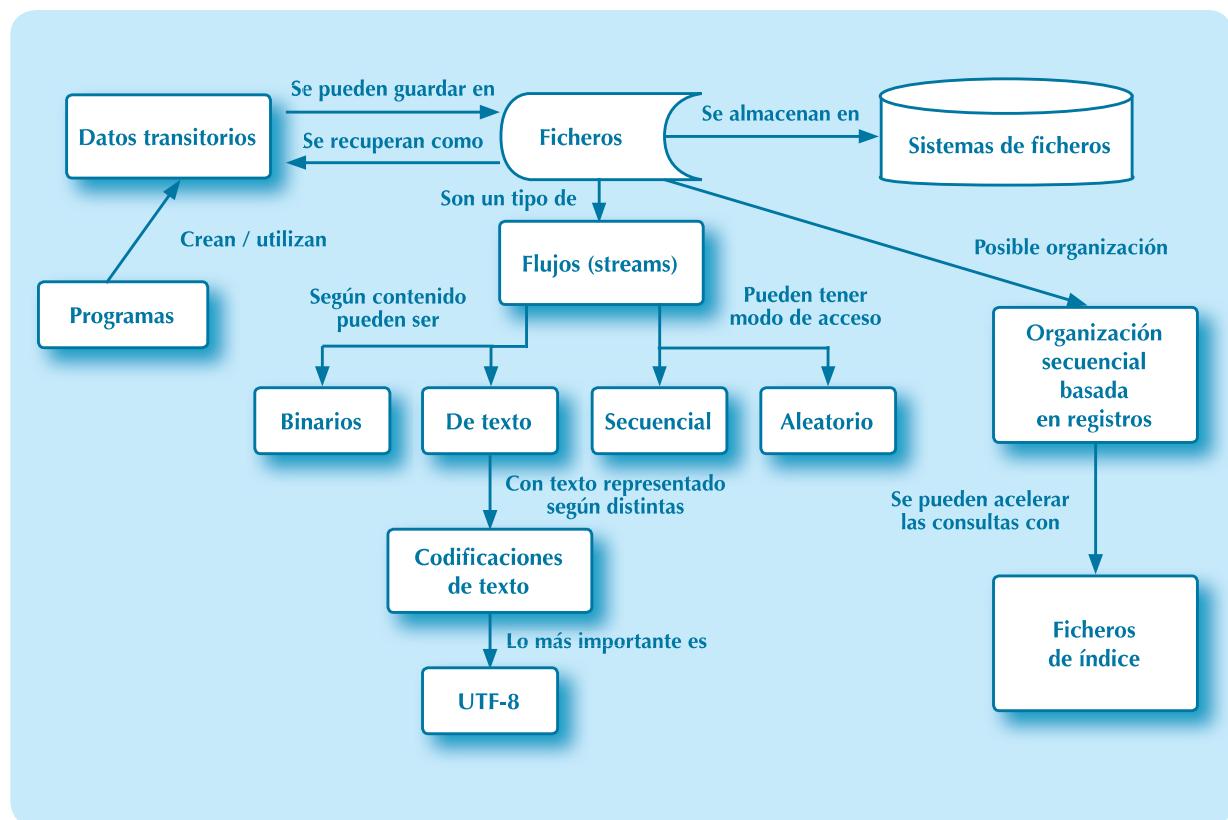


Ficheros

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 (véase figura 1.3).

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.

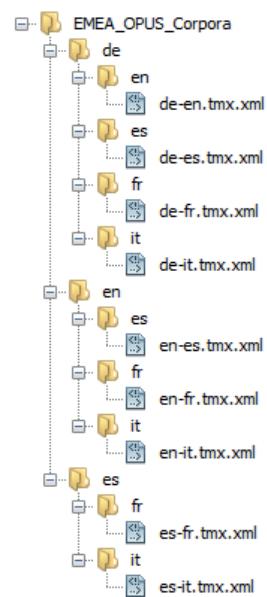


Figura 2.1
Ficheros
en una jerarquía
de directorios

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.

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.

Recurso digital



En el anexo web 2.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre la codificación del contenido de ficheros.

RECUERDA

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
```

2.4. 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`.

Recurso web

www

Se recomienda consultar en <http://docs.oracle.com/javase/8/docs/api> la documentación de Java SE8 para más información acerca de cualquier clase utilizada en este capítulo. 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.

CUADRO 2.1

Métodos de la clase `File`

Categoría	Modificador/tipo	Método(s)	Funcionalidad
Constructor		<code>File(String ruta)</code>	Crea objeto <code>File</code> para la ruta indicada, que puede corresponder a un directorio o a un fichero.
Consulta de propiedades	<code>boolean</code>	<code>canRead()</code> <code>canWrite()</code> <code>canExecute()</code>	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, <code>canExecute()</code> significa que se puede establecer como directorio actual.
	<code>boolean</code>	<code>exists()</code>	Comprueba si el fichero o directorio existe.
	<code>boolean</code>	<code>isDirectory()</code> <code>isFile()</code>	Comprueban si se trata de un directorio o de un fichero.
	<code>long</code>	<code>length()</code>	Devuelve longitud del fichero.
	<code>File</code>	<code>getParent()</code> <code>getParentFile()</code>	Devuelven el directorio padre.
	<code>String</code>	<code>getName()</code>	Devuelve nombre del fichero.
Enumeración	<code>String[]</code>	<code>list()</code>	Devuelve un array con los nombres de los directorios y ficheros dentro del directorio.
	<code>File[]</code>	<code>listFiles()</code>	Devuelve un array con los directorios y ficheros dentro del directorio.
Creación, borrado y renombrado	<code>boolean</code>	<code>createNewFile()</code>	Crea nuevo fichero.
	<code>static File</code>	<code>createTempFile()</code>	Crea nuevo fichero temporal y devuelve objeto de tipo <code>File</code> asociado, para poder trabajar con él.
	<code>boolean</code>	<code>delete()</code>	Borra fichero o directorio.
	<code>boolean</code>	<code>renameTo()</code>	Renombra fichero o directorio.
	<code>boolean</code>	<code>mkdir()</code>	Crea un directorio.
Otras	<code>java.nio.file.Path</code>	<code>toPath()</code>	Devuelve un objeto que permite acceder a información y funcionalidad adicional proporcionada por el paquete <code>java.nio</code> .

El siguiente programa de ejemplo muestra por defecto un listado de los ficheros y directorios que contiene el directorio desde el que se ejecuta el programa. Pero si se le pasa la ruta de un directorio o fichero, muestra información acerca de él y, si se trata de un directorio, muestra los ficheros y directorios que contiene.

```
// Uso de la clase File para mostrar información de ficheros y directorios
package listadodirectorio;
import java.io.File;
public class ListadoDirectorio {
    public static void main(String[] args) {
        String ruta=(".");
        if(args.length>=1) ruta=args[0];
        File fich=new File(ruta);
        if(!fich.exists()) {
            System.out.println("No existe el fichero o directorio ("+ruta+").");
        }
        else {
            if(fich.isFile()) {
                System.out.println(ruta+" es un fichero.");
            }
            else {
                System.out.println(ruta+" es un directorio. Contenidos: ");
                File[] ficheros=fich.listFiles(); // Ojo, ficheros o directorios
                for(File f: ficheros) {
                    String textoDescr=f.isDirectory() ? "/" :
                        f.isFile() ? "_" : "?";
                    System.out.println("(" +textoDescr+ ") "+f.getName());
                }
            }
        }
    }
}
```



PARA SABER MÁS

Se pueden pasar argumentos desde la línea de comandos a cualquier programa. Cualquier programa Java debe tener un método `main(String args [])` que se invoque en el momento de ejecutar el programa. El parámetro `String args[]` proporciona los argumentos de línea de comandos. Para pasar argumentos de línea de comando a un programa que se está desarrollando utilizando un IDE (entorno integrado de desarrollo), lo más cómodo suele ser utilizar las opciones que este IDE proporcione para ello dentro de su interfaz de usuario.



Actividad propuesta 2.1

Modifica el programa anterior para que muestre más información acerca de cada fichero y directorio, al menos el tamaño (si es un fichero), los permisos de los que se dispone sobre el fichero o

directorio, y la fecha de última modificación (en cualquier formato). Los permisos hay que mostrarlos en el formato habitual de Linux, a saber, con tres caracteres seguidos: una *r* si hay permiso para lectura o un guion si no lo hay; una *w* si hay permiso para escritura o un guion si no lo hay; y una *x* si hay permiso para ejecución, en el caso de que se trate de un fichero, o para entrar en el directorio, en el caso de que se trate de un directorio, o un guion si no lo hay.

2.5. Gestión de excepciones en Java

Antes de seguir avanzando con el contenido del capítulo, 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 este libro.

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, como se muestra en el siguiente ejemplo.

```
// Excepción no gestionada durante la ejecución de un programa.

package excepcionesdivporcero;

public class ExcepcionesDivPorCero {

    public int divide(int a, int b) {
        return a/b;
    }

    public static void main(String[] args) {
        int a,b;
        a=5; b=2; System.out.println(a+"/"+b+"="+a/b);
        b=0; System.out.println(a+"/"+b+"="+a/b);
        b=3; System.out.println(a+"/"+b+"="+a/b);
    }
}
```

Al ejecutar este programa se obtendrá algo similar a lo siguiente:

```
5/2=2
Exception in thread "main" java.lang.ArithmaticException: / by zero at excepcionesdivporcero.
ExcepcionesDivPorCero.main(ExcepcionesDivPorCero.java:25)
```

2.5.1. Captura y gestión de excepciones

Cuando tiene lugar una excepción no gestionada, como con el programa anterior, aparte de mostrarse un mensaje de error, se aborta la ejecución del programa. Por ese motivo, el programa anterior no muestra el resultado de la última división entera. Cualquier fragmento de programa que pueda generar una excepción debería capturarlas y gestionarlas.

La salida del programa anterior indica el tipo de excepción que se ha producido, a saber, **ArithmaticException**. Se puede capturar esta excepción con un sencillo bloque **try {} catch {}** y gestionarla, con lo que el programa anterior podría quedar así:

```
// Excepción gestionada durante la ejecución de un programa.

package excepcionesdivporcerogest;

public class ExcepcionesDivPorCeroGest {

    public int divide(int a, int b) {
        return a / b;
    }

    public static void main(String[] args) {
        int a, b;
        a = 5; b = 2;
        try {
            System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmetricException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 0; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmetricException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 3; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmetricException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
    }
}
```

Al ejecutar este programa, se captura y gestiona cualquier excepción que se pueda producir, de manera que se muestra un mensaje apropiado y se continúa con la ejecución. La salida del anterior programa sería la siguiente:

```
5/2=2
Error al dividir: 5/0
5/3=1
```

RECUERDA

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



Actividad propuesta 2.2

¿Cómo cambiaría la funcionalidad del programa anterior si, en lugar de haber un bloque `try ... catch ...` para cada división, hubiera uno solo para las tres divisiones?

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.

CUADRO 2.2**Métodos de la clase `Exception`**

Modificador/tipo	Método(s)	Funcionalidad
<code>void</code>	<code>printStackTrace()</code>	Muestra información técnica muy detallada acerca de la excepción y el contexto en que se produjo. Lo hace en la salida de error, <code>System.err</code> . 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.
<code>String</code>	<code>getMessage()</code>	Proporciona un mensaje detallado acerca de la excepción.
<code>String</code>	<code>getLocalizedMessage()</code>	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>)

El siguiente programa rellena un *array* con números y después realiza un cálculo aritmético para cada uno con ellos y muestra la segunda cifra del resultado. Esto no es realmente muy útil, como no sea para mostrar cómo se puede hacer saltar excepciones de diversos tipos, capturarlas y gestionarlas por separado. Solo para un elemento del *array* se realiza el cálculo sin que salte ninguna excepción.

```

// Gestión diferenciada de distintos tipos de excepciones

package excepcionesdiversas;

public class ExcepcionesDiversas {

    public static void main(String[] args) {
        // Rellenar array con números variados
        int nums[][] = new int[2][3];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                nums[i][j] = i + j;
            }
        }
        // Realizar cálculo para cada posición del array.
        // Se producen excepciones de diversos tipos.
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                try {
                    System.out.print("Segunda cifra de 5*nums["+i+"]"
                        +"["+j+"]/"+"+"+j+": ");
                    System.out.println(String.valueOf(5 * nums[i][j] /
                        j).charAt(1));
                } catch(ArithméticaException e) {
                    System.out.println("ERROR: aritmético 5*"+nums[i][j]+"/"+j);
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.out.println("ERROR: No existe nums["+i+"]["+j+"]");
                } catch (Exception e) {
                    System.out.println("ERROR: de otro tipo al calcular segunda
                        cifra de: 5*"+nums[i][j]+"/"+j);
                    System.out.println();
                    e.printStackTrace();
                }
            }
        }
    }
}

```

Para poder interpretar más fácilmente la salida de este programa, todos los mensajes de error se han dirigido hacia `System.out` y no `System.err`. Su salida sería:

```

Segunda cifra de 5*nums[0][0]/0: ERROR: aritmético 5*0/0
Segunda cifra de 5*nums[0][1]/1: ERROR: de otro tipo al calcular segunda cifra de: 5*1/1
java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[0][2]/2: ERROR: de otro tipo al calcular segunda cifra de: 5*2/2
java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[1][0]/0: ERROR: aritmético 5*1/0
Segunda cifra de 5*nums[1][1]/1: 0
Segunda cifra de 5*nums[1][2]/2: ERROR: de otro tipo al calcular segunda cifra de: 5*3/2

```

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 1
at java.lang.String.charAt(String.java:658)
at excepcionesdiversas.ExcepcionesDiversas.main(ExcepcionesDiversas.java:31)
Segunda cifra de 5*nums[2][0]/0: ERROR: No existe nums[2][0]
Segunda cifra de 5*nums[2][1]/1: ERROR: No existe nums[2][1]
Segunda cifra de 5*nums[2][2]/2: ERROR: No existe nums[2][2]
```

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.`

La siguiente clase tiene un método que crea un fichero temporal con un nombre que empieza por un prefijo dado, y escribe en él un carácter dado, un número dado de veces. Durante este proceso puede saltar la excepción `IOException` en varias ocasiones, pero no se quiere gestionarla en el método. Por ello se incluye `throws IOException` en su declaración. Por lo demás, el método es sencillo y, en cualquier caso, lo importante es comprender la gestión de excepciones. Las clases y procedimientos utilizados se verán más adelante en este capítulo.

```
// Declaración de excepciones lanzadas por método de clase con throws
package excepcionesconthrows;

import java.io.File;
import java.io.IOException;
import java.io.FileWriter;

public class ExcepcionesConThrows {

    public File creaFicheroTempConCar(String prefNomFich, char car, int numRep) throws IOException {
        File f = File.createTempFile(prefNomFich, "");
        FileWriter fw = new FileWriter(f);
        for (int i = 0; i < numRep; i++) fw.write(car);
        fw.close();
        return f;
    }

    public static void main(String[] args) {
        try {
            File ft = new ExcepcionesConThrows().
                creaFicheroTempConCar("AAAA_", 'A', 20);
            System.out.println("Creado fichero: " + ft.getAbsolutePath());
            ft.delete();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

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()
}

```

Recurso digital



En el anexo web 2.2 encontrarás un ejemplo que simula un programa que abre para lectura dos ficheros: f1.dat y f2.dat, y que crea dos ficheros temporales: f1.info.tmp y f2.info.tmp, que borra al final.

El anexo está disponible en www.sintesis.com y es accesible con el código indicado en la primera página del libro.

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.

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 volverlos 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.

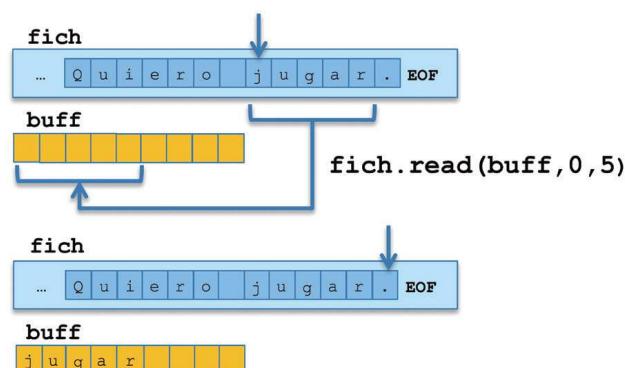


Figura 2.2
Lectura desde fichero

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. 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.

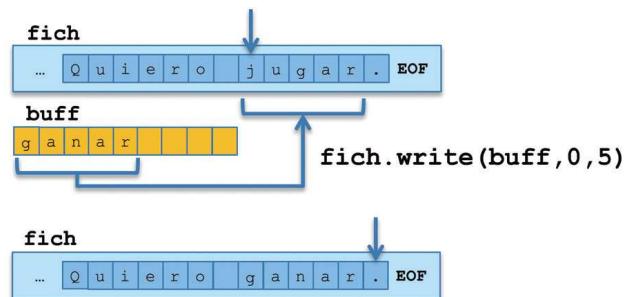
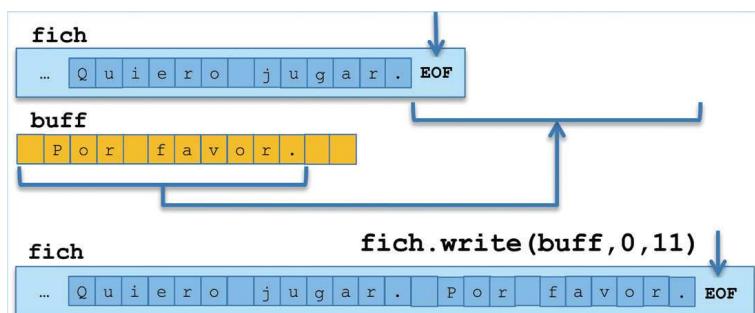


Figura 2.3

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

Figura 2.4
Escríbete a 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.

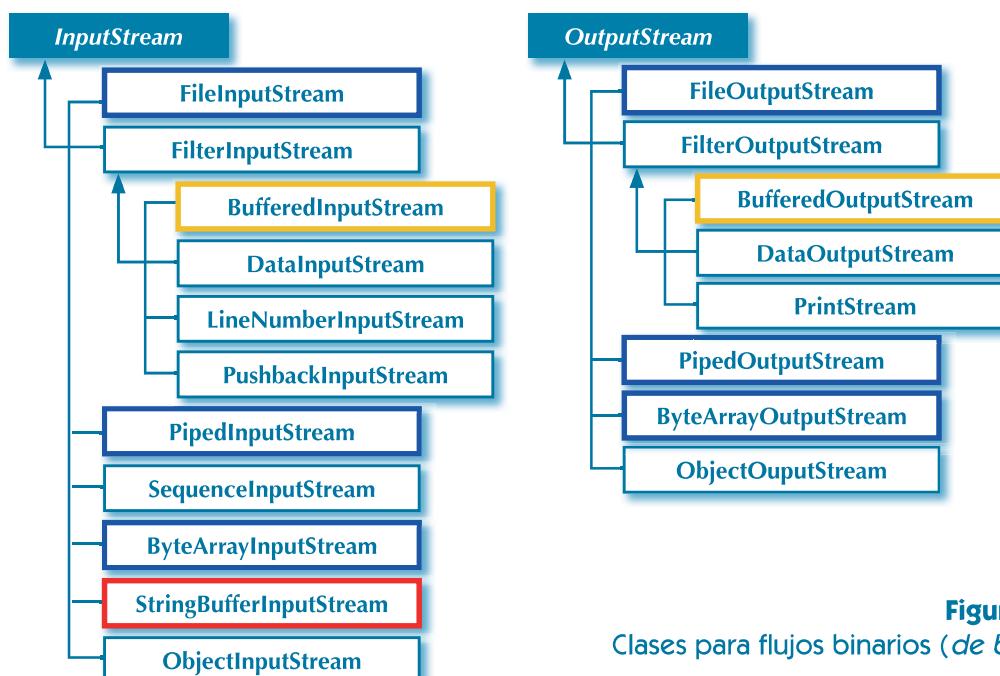


Figura 2.5
Clases para flujos binarios (*de bytes*)



TOMA NOTA

La clase `StringBufferInputStream` está obsoleta (*deprecated* según la terminología de Java). En su lugar habría que usar `StringReader`.

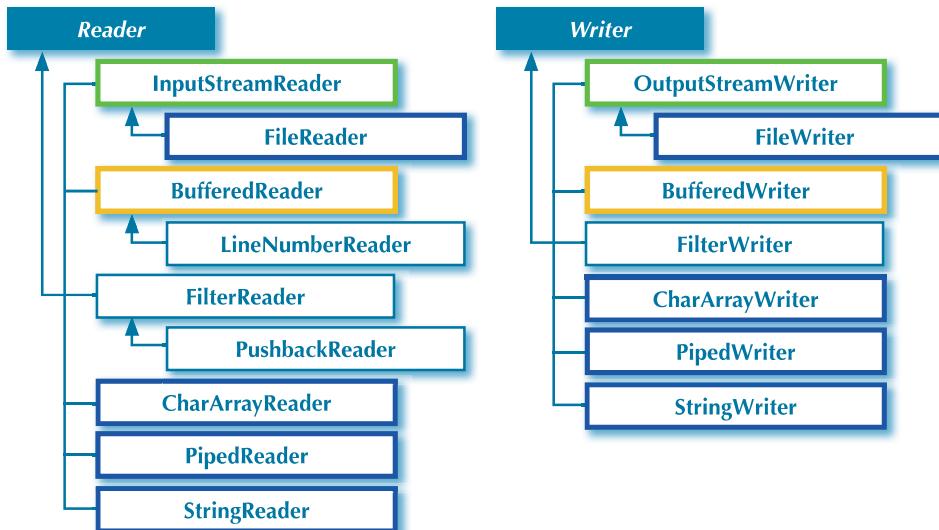


Figura 2.6
Clases para flujos de texto

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.

CUADRO 2.3

Clases básicas para entrada y salida a flujos

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

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.



Figura 2.7. Lectura y escritura en ficheros binarios



Figura 2.8. Lectura y escritura en ficheros de texto con la codificación por defecto

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).

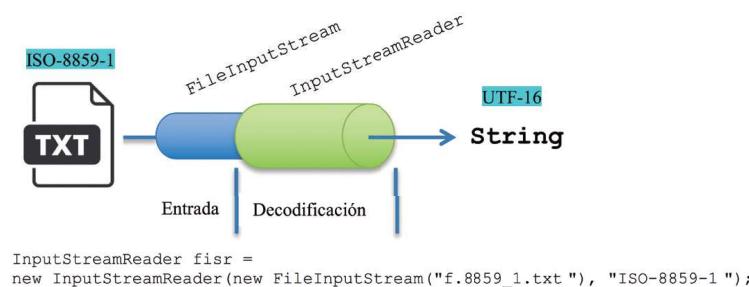


Figura 2.9.
Lectura desde
ficheros de
texto con
codificación
distinta a la
codificación
por defecto

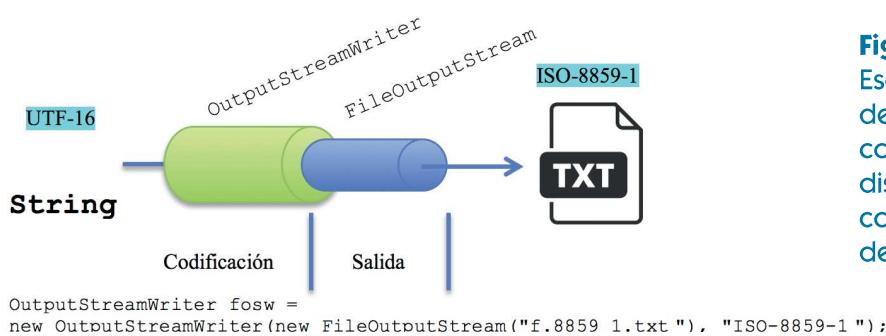


Figura 2.10.
Escritura a ficheros
de texto con
codificación
distinta a la
codificación por
defecto

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*.

CUADRO 2.4

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*:

CUADRO 2.5

Flujo sin buffering y con buffering

Flujo sin buffering	Flujo con buffering
<code>new FileInputStream("f.bin")</code>	<code>new BufferedInputStream(new FileInputStream("f.bin"))</code>
<code>new FileOutputStream("f.bin")</code>	<code>new BufferedOutputStream(new FileOutputStream("f.bin"))</code>
	[.../...]

CUADRO 2.5 (CONT.)

<code>new FileReader("f.txt")</code>	<code>new BufferedReader(new FileReader("f.txt"))</code>
<code>new FileWriter("f.txt")</code>	<code>new BufferedWriter(new FileWriter("f.txt"))</code>

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

CUADRO 2.6**Métodos para lectura y escritura de líneas en clases para buffering con ficheros de texto**

Clase	Método	Funcionalidad
<code>BufferedReader</code>	<code>String readLine()</code>	Lee hasta el final de la línea actual.
<code>BufferedWriter</code>	<code>void newLine()</code>	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 <code>readLine()</code> de <code>BufferedReader</code> 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`.

CUADRO 2.7**Métodos para lectura de las clases para gestión de flujos de entrada**

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

Como ejemplo, el siguiente programa muestra los contenidos de un fichero de texto línea a línea, numerando las líneas. Para leer líneas de texto se usa el método `readLine()` de la clase `BufferedReader`. En este programa, y en todos a partir de ahora, se utilizarán bloques `try` con recursos para crear distintos tipos de flujos (*stream*), con lo que `close()` se ejecutará automáticamente al final.

```
// Uso de readLine() de BufferedReader

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class EscribeConNumeroDeLineas {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Indicar por favor nombre de fichero.");
            return;
        }
        String nomFich = args[0];

        try(BufferedReader fbr = new BufferedReader(new FileReader(nomFich))) {
            int i = 0;
            String linea = fbr.readLine();
            while (linea != null) {
                System.out.format("[%5d] %s", i++, linea);
                System.out.println();
                linea = fbr.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("No existe fichero " + nomFich);
        } catch (IOException e) {
            System.out.println("Error de E/S: " + e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Actividades propuestas



- 2.3.** Crea un programa que busque un texto dado en un fichero de texto, y que muestre para cada aparición la línea y la columna. Se recomienda leer el fichero línea a línea y, dentro de cada línea, buscar las apariciones del texto utilizando un método apropiado de la clase `String`. Se puede consultar la documentación de dicha clase en la API de Java (<http://docs.oracle.com/javase/8/docs/api>).
- 2.4.** Crea un programa que, a partir de un fichero de texto codificado en UTF-8, genere un fichero de texto codificado en ISO-8859-1 y otro en UTF-16. El fichero codificado en UTF-8 debe crearse con un editor de texto, y debe incluir al menos vocales acentuadas. Puedes leer el fichero línea a línea con `readLine()`. Para generar el fichero de salida, puedes utilizar un `BufferedWriter` (para escribir línea a línea) construido sobre un `OutputStreamWriter` (para recodificar el texto) construido sobre un `FileOutputStream` (para escribir a un fichero). Busca una manera de verificar la codificación de los

ficheros de texto en el sistema operativo que estés utilizando mediante algún comando del sistema operativo o algún programa de utilidad. Puedes utilizar los programas de ejemplo para volcado binario para verificar qué caracteres se codifican de manera distinta. Puedes crear otro programa que haga la conversión inversa, para comprobar que se vuelve a obtener un fichero igual al inicial, utilizando las clases `InputStreamReader` y `FileInputStream`.

Ahora un ejemplo con flujos binarios. El siguiente programa hace un volcado binario de un fichero indicado desde línea de comandos. Los contenidos del fichero se leen en bloques de 32 *bytes*, y el contenido de cada bloque se escribe en una línea de texto. Los *bytes* se escriben en hexadecimal (base 16) y, por tanto, cada *byte* se escribe utilizando dos caracteres. El programa muestra como máximo los primeros 2 *kilobytes* (`MAX_BYTES=2048`). Por supuesto, este programa se puede utilizar tanto con ficheros binarios como con ficheros de texto. Hacer notar que esta clase permite hacer el volcado binario de un `InputStream`, y un `FileInputStream` es un caso particular. Siempre que sea posible, debemos hacer que las clases que desarrollemos funcionen con *streams* en general, y no solo con ficheros en particular.

```
// Volcado hexadecimal de un fichero con FileInputStream
package volcadobinario;

import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class VolcadoBinario {

    static int TAM_FILA=32;
    static int MAX_BYTES=2048;
    InputStream is=null;

    public VolcadoBinario(InputStream is) {
        this.is=is;
    }

    public void volcar() throws IOException {
        byte buffer[]=new byte[TAM_FILA];
        int bytesLeidos;
        int offset=0;
        do { bytesLeidos=is.read(buffer);
            System.out.format("[%5d]", offset);
            for(int i=0; i<bytesLeidos; i++) {
                System.out.format(" %2x", buffer[i]);
            }
            offset+=bytesLeidos;
            System.out.println();
        } while (bytesLeidos==TAM_FILA && offset<MAX_BYTES);
    }

    public static void main(String[] args) {
        if(args.length<1) {
            System.out.println("No se ha indicado ningún fichero");
            return;
        }
    }
}
```

```
String nomFich=args[0];
try (FileInputStream fis = new FileInputStream(nomFich)) {
    VolcadoBinario vb = new VolcadoBinario(fis);
    vb.volcar();
}
catch(FileNotFoundException e) {
    System.err.println("ERROR: no existe fichero "+nomFich);
}
catch(IOException e) {
    System.err.println("ERROR de E/S: "+e.getMessage());
}
catch(Exception e) {
    e.printStackTrace();
}
}
```

Actividad propuesta 2.5



Modifica la clase `VolcadoBinario` para que pueda hacer el volcado a cualquier `PrintStream`, en lugar de siempre a `System.out`. Modifica el método `main()` para que realice el volcado hacia un fichero.

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)
```

CUADRO 2.8

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)
	void write(String str, int offset, int longitud)

[... / ...]

CUADRO 2.8 (CONT.)

```

    Writer append(char c)
    Writer append(CharSequence csq)
    Writer append(CharSequence csq, int offset, int
                  longitud)

```

BufferedWriter

```

void newLine()

```

El siguiente programa escribe un texto en un fichero. Despu s lo cierra y lo vuelve a abrir en modo *append* para a adir nuevos contenidos al final. A menos que el fichero ya exista, en cuyo caso no hace nada. Si se hicieran estas mismas operaciones sobre un fichero existente, se perder n los contenidos del fichero. Se a aden saltos de l nea con **newLine()**.

```

// A adir contenidos al final de un fichero de texto

package escribeenflujosalida;

import java.io.File;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class EscribeEnFlujoSalida {

    public static void main(String[] args) {

        String nomFichero="f_texto.txt";
        File f=new File(nomFichero);
        if(f.exists()) {
            System.out.println("Fichero "+nomFichero+" ya existe. No se hace
                               nada");
            return;
        }

        try {
            BufferedWriter bfw=new BufferedWriter(new FileWriter(f));
            bfw.write(" Este es un fichero de texto. ");
            bfw.newLine();
            bfw.write(" quiz  no est  del todo bien.");
            bfw.newLine();
            bfw.close();
            bfw=new BufferedWriter(new FileWriter(f, true));
            bfw.write(" Pero se puede arreglar.");
            bfw.newLine();
            bfw.close();
        }
        catch(IOException e) {
            System.out.println(e.getMessage());
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

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`.

El siguiente programa realiza diversos cambios en los contenidos de un fichero de texto tales como eliminar secuencias de espacios al principio de línea, sustituir secuencias de espacios en otros lugares por un solo espacio, y hacer que todas las líneas empiecen por mayúsculas. Se puede probar con el fichero generado por el programa anterior.

Para las transformaciones en el texto se utiliza funcionalidad de la clase `Character`. Lo más importante no es entender en detalle lo que hace dentro del bucle `while` para cada línea, sino la técnica que emplea para modificar los contenidos de un fichero, leyendo de un `BufferedReader`, escribiendo en un `BufferedWriter`, utilizando ficheros temporales, y renombrando ficheros. Antes que nada, y por si acaso, se hace una copia del fichero en uno nuevo en cuyo nombre aparece una marca de tiempo incluyendo la fecha y hora exactas. Los nuevos contenidos del fichero se escriben en un fichero temporal. Al terminar, se borra el fichero original y se renombra el fichero temporal con el nombre del fichero original. El renombrado de ficheros consiste no solo en un cambio de nombre, sino también de ubicación, si se especifica un directorio distinto a aquel en que está ubicado el fichero. Es recomendable, y se puede ver que es muy sencillo, hacer que los programas realicen copias de seguridad de todos aquellos ficheros que vayan a modificar, al menos hasta que se hayan probado lo suficiente, después de lo cual se puede eliminar esta parte del programa.

```
// Cambio de contenidos de ficheros utilizando flujos de lectura y escritura y
// ficheros temporales

package arreglaficherotexto;

import java.io.File;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;
import java.text.SimpleDateFormat;

public class ArreglaFicheroTexto {

    public static void main(String[] args) {

        String nomFichero = "f_texto.txt";
        File f = new File(nomFichero);
        if (!f.exists()) {
            System.out.println("Fichero " + nomFichero + " no existe.");
            return;
        }
        try (BufferedReader bfr = new BufferedReader(new FileReader(f))) {
            File fTemp = File.createTempFile(nomFichero, "");
            System.out.println("Creado fich. temporal " + fTemp.
                getAbsolutePath());
            BufferedWriter bfw = new BufferedWriter(new FileWriter(fTemp));
            String linea = bfr.readLine();
            while (linea != null) { // En resumen, lee de bfr y escribe en bfw
                boolean principioLinea = true, espacios = false,
                    primerAlfab=false;
```

```

        for (int i = 0; i < linea.length(); i++) {
            char c = linea.charAt(i);
            if (Character.isWhitespace(c)) {
                if (!espacios && !principioLinea) {
                    bfw.write(c);
                }
                espacios = true;
            } else if (Character.isAlphabetic(c)) {
                if(!primerAlfab) {
                    bfw.write(Character.toUpperCase(c));
                    primerAlfab=true;
                }
                else bfw.write(c);
                espacios = false;
                principioLinea = false;
            }
        }
        bfw.newLine();
        linea = bfr.readLine();
    }
    bfw.close();
    f.renameTo( // Copia de seguridad
        new File(nomFichero+
            "."+new SimpleDateFormat("yyyyMMddHHmmss").format(new
            Date())+".bak"
        )
    );
    fTemp.renameTo(new File(nomFichero));
} catch (IOException e) {
    System.out.println(e.getMessage());
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

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 necesiten 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 períodos 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`.

CUADRO 2.9

Principales métodos de la clase `RandomAccessFile`

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

La clase desarrollada en el siguiente ejemplo permite almacenar registros con datos de clientes en un fichero de acceso aleatorio. Los datos de cada cliente se almacenan en un registro, que es una estructura de longitud fija dividida en campos de longitud fija. En este caso, los campos son DNI, nombre y código postal. El constructor de la clase toma una lista con la definición del registro. Cada elemento de la lista contiene la definición de un campo en un par <nombre, longitud>. Los valores de los campos para un registro se almacenan en un [HashMap](#), que contiene pares <nombre, valor>, cada uno de los cuales contiene el valor para un campo.

Al constructor se le proporciona un nombre de fichero. Si el fichero no existe, se crea. Si el fichero existe, se calcula el número de registros que contiene, dividiendo la longitud del fichero en *bytes* por la longitud de cada registro.

El método más interesante es [insertar\(\)](#). Tiene dos variantes. Si no se le indica la posición, añade el registro al final del fichero. Si no, en la posición que se le indique. La posición del primer registro es 0, no 1. Los textos se almacenan siempre codificados en UTF-8. Como es relativamente habitual en los métodos, no gestionan las excepciones que puede generar ([throws IOException](#)), y dejan esto para el programa principal.

```
// Almacenamiento de registros de longitud fija en fichero acceso aleatorio

package ficheroaccesoaleatorio;

import java.io.File;
import java.io.RandomAccessFile;
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import javafx.util.Pair;

public class FicheroAccesoAleatorio {

    private File f;
    private List<Pair<String, Integer>> campos;
    private long longReg;
    private long numReg = 0;

    FicheroAccesoAleatorio(String nomFich, List<Pair<String, Integer>> campos)
        throws IOException {
        this.campos = campos;
        this.f = new File(nomFich);
        longReg = 0;
        for (Pair<String, Integer> campo: campos) {
            this.longReg += campo.getValue();
        }
        if(f.exists()) {
            this.numReg=f.length()/this.longReg;
        }
    }

    public longgetNumReg() {
        return numReg;
    }

    public void insertar(Map<String, String> reg) throws IOException {
        insertar(reg, this.numReg++);
    }
}
```

```

public void insertar(Map<String, String> reg, long pos) throws IOException {
    try(RandomAccessFile faa = new RandomAccessFile(f, "rws")) {
        faa.seek(pos * this.longReg);
        for (Pair<String, Integer> campo: this.campos) {
            String nomCampo=campo.getKey();
            Integer longCampo = campo.getValue();
            String valorCampo = reg.get(nomCampo);
            if (valorCampo == null) {
                valorCampo = "";
            }
            String valorCampoForm = String.format("%1$-" + longCampo + "s",
                valorCampo);
            faa.write(valorCampoForm.getBytes("UTF-8"), 0, longCampo);
        }
    }
}

public static void main(String[] args) {
    List campos = new ArrayList();
    campos.add(new Pair("DNI", 9));
    campos.add(new Pair("NOMBRE", 32));
    campos.add(new Pair("CP", 5));

    try {
        FicheroAccesoAleatorio faa = new FicheroAccesoAleatorio("fic_acceso_
            aleat.dat", campos);
        Map reg = new HashMap();
        reg.put("DNI", "56789012B");
        reg.put("NOMBRE", "SAMPER");
        reg.put("CP", "29730");
        faa.insertar(reg);
        reg.clear();
        reg.put("DNI", "89012345E");
        reg.put("NOMBRE", "ROJAS");
        faa.insertar(reg);
        reg.clear();
        reg.put("DNI", "23456789D");
        reg.put("NOMBRE", "DORCE");
        reg.put("CP", "13700");
        faa.insertar(reg);
        reg.clear();
        reg.put("DNI", "78901234X");
        reg.put("NOMBRE", "NADALES");
        reg.put("CP", "44126");
        faa.insertar(reg,1);
    } catch (IOException e) {
        System.err.println("Error de E/S: " + e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```



Actividades propuestas

- 2.6.** ¿Qué crees que pasaría si se intenta usar la clase `FicheroAccesoAleatorio` para almacenar un registro en una posición mayor que el número de registros que contiene el fichero? Compruébalo modificando el método `main()` para hacer las pruebas oportunas.
- 2.7.** Completa la clase `FicheroAccesoAleatorio` con un método que permita obtener el valor de un campo de un registro, dada la posición del registro y el nombre del campo. Por ejemplo: se podría acceder al valor del campo "Nombre" del registro situado en la posición 5 con `obtenerValorCampo(4, "Nombre")`. Por coherencia con la manera en que se ha implementado el método `insertar()`, la posición del primer registro debe entenderse que es 0, no 1. `String` debe tener el mismo formato que en la línea siguiente. Se recomienda utilizar el constructor `String(byte[] bytes, Charset charset)` de `String`.

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 datos 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 en-

contrario al 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 inefficientes. 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 inefficiente 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 (véase figura 1.5).

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.



Ejercicios propuestos

Los ejercicios propuestos a continuación se centran en las organizaciones de ficheros y las operaciones para cada una de ellas: inserción, borrado, modificación de ficheros, indexación y ordenación. Para la realización de estas actividades podría ser necesario utilizar clases, métodos u opciones no vistos en el capítulo, por lo que se recomienda consultar la documentación de Java SE 8 (<http://docs.oracle.com/javase/8/docs/api/>).

Nota: cuando se desarrolle una clase, debe crearse un método `main()` que pruebe suficientemente la funcionalidad de la clase. Si se piden modificaciones de una clase existente, las pruebas deberán incidir sobre la funcionalidad añadida o modificada y sobre cualquier otra funcionalidad que haya podido verse afectada por el cambio.

1. Crea una clase que implemente las operaciones para añadir, recuperar y modificar registros de un fichero con organización secuencial. Los nuevos registros se añadirán siempre al final. El fichero debe tener un campo clave, de manera que no se permita que existan dos registros con el mismo valor para el campo clave. Para recuperar un registro se proporcionará el valor de su campo clave. Para modificarlo se proporcionará el valor del campo clave para buscar el registro que se va a modificar, el nombre de un campo y el nuevo valor de ese campo. Ten en cuenta que el hecho de que el registro tenga organización secuencial no significa que no se pueda utilizar acceso aleatorio para él, si esto supone un beneficio para realizar algunas operaciones puedes implementar la clase con una estructura fija de fichero para que sea más sencillo. Por ejemplo: campos DNI y nombre de cliente, campo clave DNI.
2. Basándote en el ejercicio anterior, crea una nueva clase, más genérica, que permita definir la estructura del registro al crear un nuevo fichero, como se hace en un programa de ejemplo para ficheros de acceso aleatorio. Todo funcionará igual, solo que la clase valdrá para cualquier estructura de registro, en lugar de para una estructura fija.
3. Añade un método para borrar registros a la clase anterior. Pero los registros realmente no se borrarán, sino que se marcarán como borrados. Se puede, sencillamente, poner un carácter especial al principio del registro. Cuando se inserte un registro, se insertará, como antes, al final del fichero. Se pueden explorar otras posibilidades para marcar los registros como borrados. Por ejemplo, asignar todo espacios o asignar un valor cero a todos los bytes.
4. Añade a la clase anterior un método para compactar el fichero, es decir, para eliminar los registros marcados como borrados. Habrá que utilizar un fichero temporal para construir el nuevo fichero y, finalmente, sustituir el antiguo por el nuevo.
5. Crea una clase que implemente un fichero secuencial que contenga registros de longitud fija, compuestos por campos de longitud fija, y que siempre esté ordenado por el valor del campo clave. Cuando se inserta un nuevo registro hay que hacerlo en la posición que deba ocupar en el fichero para preservar su ordenación. Se sugiere generar el nuevo fichero en un fichero temporal y sustituir el antiguo por este al final. En el fichero temporal se irán copiando los registros del fichero hasta llegar a uno con un valor mayor para el campo clave. Entonces, se insertará el nuevo registro en el fichero temporal, y después se copiará el resto de los registros en el fichero temporal.

Por último, el fichero actual se sustituirá por el fichero temporal. En este capítulo se han visto programas de ejemplo que utilizan ficheros temporales como ayuda para modificar los contenidos de un fichero.

6. Crea una clase que implemente operaciones para añadir, buscar, borrar y modificar registros de un fichero con organización secuencial indexada. Existirá un único índice para el campo clave del fichero. El índice será un fichero adicional cuyo nombre indique el fichero principal (de datos) y el campo de indexación (que, en este caso, y según se ha dicho, es el campo clave). El índice lo creará el constructor de la clase, junto con el fichero principal. Los registros borrados no se deben borrar, sino marcarse como borrados tanto en el fichero principal como en el fichero de índice. En este último se podría poner un número negativo como posición para indicar que el registro está borrado. Los nuevos registros se añadirán siempre al final del fichero principal. La operación de inserción debe recomponer el índice, para ello hay que insertar una nueva entrada en el lugar apropiado. El índice no es más que un tipo especial de fichero secuencial ordenado, y el fichero principal un fichero secuencial no ordenado, por lo que se pueden utilizar las clases desarrolladas en ejercicios anteriores.