Empezar a Programar Usando Java

Natividad Prieto
Francisco Marqués
Isabel Galiano
Jorge González
Carlos Martínez-Hinarejos
Javier Piris
Assumpció Casanova
Marisa Llorens
Jon Ander Gómez
Carlos Herrero
Germán Moltó

Capítulo 15

Entrada y salida: ficheros y flujos

Este capítulo introduce los conceptos necesarios y explica las principales clases Java que permiten leer, escribir y gestionar ficheros. Mediante las operaciones relativas a la Entrada/Salida (E/S) básica, los programas en Java pueden crear y manipular ficheros, bien para escribir información en ellos, bien para leer de los mismos.

Por supuesto, la E/S básica es uno de los elementos fundamentales relativos a los lenguajes de programación. En efecto, los datos de las variables definidas en un programa desaparecen cuando éste finaliza. Por lo tanto, si se desea que los datos generados por un programa puedan estar disponibles para ejecuciones posteriores

(o para otros programas), una forma sencilla de hacerlo es almacenándolos en un fichero.

Un fichero es una secuencia de bytes guardados en un dispositivo de almacenamiento secundario (disco duro, USB stick, etc.). Los ficheros se identifican por un nombre y, por lo general, también llevan asociada una extensión que, en algunos sistemas operativos, puede ayudar a identificar el tipo de contenido del mismo. Por ejemplo, resulta lógico pensar que un fichero llamado *canciones.txt* contenga información de texto aunque, en realidad, la extensión de un fichero no determina necesariamente el contenido del mismo.

Es importante destacar que el manejo de ficheros es tan solo un caso particular de la gestión de E/S en Java, donde aparece el concepto de flujo (*stream*). Como ya se vió en el capítulo 6, un flujo representa una secuencia de bytes que se reciben desde una fuente (p.e., teclado, fichero, red, etc.) y se dirigen a un destino (p.e., pantalla, fichero, dispositivo, etc.). Si el flujo de datos se dirige al programa, es decir, representa una entrada de datos para el mismo, entonces se habla de flujo de entrada (al programa). Por el contrario, si el flujo de datos parte del programa,

es decir, se trata de una salida de datos, entonces se habla de flujo de salida (del programa). La noción de *flujo* es muy amplia y mediante distintas extensiones de la misma se tratan en el lenguaje Java muchos aspectos, como los relativos a conexiones remotas con otros ordenadores, manipulación de información multimedia así como E/S estándar y con ficheros.

También hay que destacar que en este capítulo se van a abordar principalmente los ficheros de acceso secuencial, en los cuales la lectura de los datos típicamente se realiza en el mismo orden en el que éstos se han guardado. Por ejemplo, una aplicación de agenda telefónica precisaría almacenar sus datos en un fichero para que estén disponibles siempre que se ejecute la aplicación. La figura 15.1 muestra una posible organización del contenido de dicho fichero, que consta de tantos registros como entradas haya en la agenda y donde cada registro incluye varios campos (nombre, dirección y teléfono). Por lo tanto, la aplicación guarda estos datos en el fichero en un determinado orden y, posteriormente, la lectura se realiza en el mismo orden en el que los datos fueron escritos. Los ficheros secuenciales son posiblemente los más comunes y por eso este capítulo se centra en ellos. No obstante, la sección 15.3.3 aborda las posibilidades que ofrecen los ficheros de acceso aleatorio.

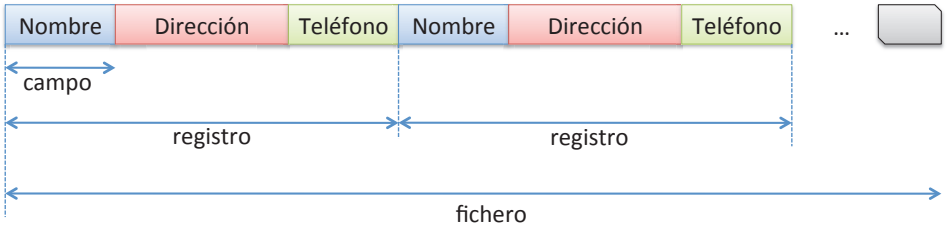

Figura 15.1: Ejemplo de fichero de acceso secuencial que incluye varios registros de una agenda telefónica.

En general, los datos pueden estar almacenados tanto en un fichero de texto como en uno binario, por lo que es muy importante conocer las diferencias, ventajas e inconvenientes de cada tipo. Además, la forma en la que se accede a los ficheros en Java es diferente según se trate de un fichero de texto o uno binario.

Ficheros de texto y binarios

Un *fichero de texto* consiste en una secuencia de caracteres almacenada mediante un esquema de codificación que suele ser ASCII (o UTF-8, que incluye todos los caracteres soportados por ASCII más otros caracteres usados por otras lenguas). Es común, por tanto, encontrar en la literatura el término fichero ASCII para referirse a un fichero de texto plano. Un fichero de este tipo es directamente interpretable por una persona, que puede ver el contenido del mismo y los caracteres que en él

hay escritos. Los ficheros de texto son muy portables, lo que significa que pueden ser leídos desde cualquier otro equipo con diferente sistema operativo o desde un lenguaje de programación o editor distinto al empleado para construir el fichero.

Por el contrario, un *fichero binario* consiste en una secuencia de bytes y, por lo tanto, no es directamente interpretable por una persona. Los ficheros binarios se escriben desde un programa, a través de las operaciones de E/S soportadas por un lenguaje de programación. En general, los ficheros binarios no son demasiado portables ya que dependen de la arquitectura de la máquina y del lenguaje de programación empleado para escribir el fichero, para poder interpretar correctamente la información del fichero. Afortunadamente, un fichero binario escrito desde Java puede ser leído desde cualquier otro equipo que tenga un sistema operativo diferente pero, eso sí, la lectura debe ser realizada desde un programa Java.

La principal ventaja de los ficheros de texto, por lo tanto, es la gran portabilidad. No obstante, estos ficheros requieren más tamaño que un fichero binario para representar la misma información[1]. Por otra parte, las operaciones sobre ficheros binarios resultan más eficientes que sobre ficheros de texto. Finalmente, es importante recordar que, mientras que un fichero de texto es accesible mediante cualquier editor de textos, un fichero binario requiere la construcción de un programa para su lectura.

15.1 La clase `File`

La clase `java.io.File` permite abstraer el concepto de fichero y directorio desde el lenguaje de programación. Ofrece métodos para interaccionar con el sistema de archivos mediante los que se pueden obtener las principales propiedades de un fichero o directorio (nombre, ruta, si es modificable o no, etc.), así como algunas operaciones sobre los mismos (renombrar, borrar, etc.). En esta sección, se utiliza de forma indistinta el concepto de fichero o directorio ya que la clase `File` los trata de igual manera.

La figura 15.2 muestra un ejemplo de uso de la clase `File`. En primer lugar se procede a la construcción del objeto de tipo `File` (línea 12). Para ello, hay que indicarle una ruta absoluta (como en el ejemplo), que identifica de forma inequívoca un fichero en el sistema de archivos, una ruta relativa al directorio de trabajo, o bien directamente un nombre de fichero. En los dos últimos casos, se asume como directorio de trabajo aquel desde el que se ejecuta el programa.

La creación de un objeto `File` no implica la construcción de un fichero en el sistema de archivos. De hecho, la ruta al fichero especificado ni siquiera tiene

[1]Por ejemplo, un entero de 10 dígitos en un fichero de texto ocuparía 10 bytes (asumiendo codificación ASCII de 1 byte/carácter), mientras que en un fichero binario el mismo número ocuparía los 4 bytes correspondientes a almacenar un tipo `int`.

que corresponder con un fichero que realmente exista. La clase `File` proporciona métodos para verificar posteriormente si realmente existe un fichero en dicha ruta.

```
1  import java.io.File;
2  /**
3   * Clase TestFile: uso de la clase File.
4   * @author Libro IIP-PRG
5   * @version 2016
6   */
7  public class TestFile {
8      /** Método principal.
9       *  @param args String[], argumentos del programa.
10      */
11     public static void main(String[] args) {
12         File f = new File("/tmp/file.txt");
13         if (f.exists()) {
14             System.out.println("El fichero existe!");
15             System.out.println("getName(): " + f.getName());
16             System.out.println("getParent(): " + f.getParent());
17             System.out.println("length(): " + f.length());
18         } else { System.err.println("El fichero NO existe!"); }
19     }
20 }
```

Figura 15.2: Ejemplo de uso de la clase `File`.

Conviene destacar que especificar una ruta a un fichero en Windows mediante un `String`, puede hacerse tanto mediante el uso de la barra estándar del Unix, como utilizando doble contrabarra (debido al uso especial del carácter '\' en Java). A continuación se muestran los dos métodos alternativos, empleados para definir el acceso a un fichero en un sistema Windows.

```
File f = new File("C:/Users/lucas/file.txt");
File f = new File("C:\\Users\\lucas\\file.txt");
```

Adicionalmente, para la definición de rutas de acceso a ficheros existen en Java, definidos en la clase `File`, los atributos estáticos finales de tipo `char`:

- `File.separatorChar`,
- `File.pathSeparatorChar`

que contienen durante la ejecución de un programa el carácter utilizado en el sistema operativo en uso (Linux, OS X, Windows. etc.) bien como separador de ficheros bien como separador en la expresión de rutas de acceso. Pueden utilizarse en la formación de los nombres de ficheros y/o rutas, de forma que se independice el programa que se construya del sistema operativo que se vaya a utilizar.

Volviendo con el ejemplo, tras la construcción del objeto `File`, se comprueba si realmente el fichero existe en el sistema de archivos mediante el método `exists()` (línea 13). A continuación se utilizan algunos métodos para obtener el nombre del fichero, su directorio padre y la longitud del mismo (en bytes), respectivamente (líneas 15-17).

Asumiendo que el fichero */tmp/file.txt* del ejemplo existe y contiene la cadena "Hola", la salida del programa sería la siguiente:

Salida Estándar

```
El fichero existe!
getName(): file.txt
getParent(): /tmp
length(): 4
```

Nótese que la longitud del fichero (4 bytes) coincide con el número de caracteres de su contenido[2]. Si el fichero no hubiera existido, las únicas dos diferencias significativas serían el diferente mensaje mostrado por la salida estándar de error (línea 18) y el tamaño del fichero que sería 0 bytes. Es importante destacar que el método `length()` tan solo obtiene el valor del tamaño para ficheros, no para directorios. Calcular el tamaño de un directorio, a partir de la suma de los tamaños de sus ficheros y sus directorios, requiere típicamente el uso de un método recursivo.

La tabla 15.1 incluye un resumen de los principales constructores y métodos disponibles en la clase `File`. No obstante, se recomienda al lector consultar el *API* de Java [Ora16b] para verificar la funcionalidad completa de las clases. Concretamente, la clase `File` también permite construir directorios, crear nuevos ficheros vacíos, distinguir entre un fichero y un directorio, etc.

`public` **`File`**`(String pathname)`	Crea un nuevo `File` a partir de la ruta a un fichero (o directorio).
`public boolean` **`delete`**`()`	Intenta eliminar el fichero (o un directorio vacío). Devuelve `true` en caso de éxito.
`public boolean` **`renameTo`**`(File dest)`	Renombra el fichero al nuevo nombre especificado. Puede involucrar mover el fichero en el sistema de archivos. Devuelve `true` en caso de éxito.
`public boolean` **`exists`**`()`	Devuelve `true` si el fichero existe en el sistema de archivos.

Tabla 15.1: Clase `File`: principales constructores y métodos.

[2]En realidad el tamaño depende de la codificación utilizada para guardar el fichero. En este caso se utilizó una codificación ASCII que involucra 1 byte por carácter.

15.2 Ficheros de texto

Esta sección aborda el proceso de escritura y lectura de ficheros de texto. Como un extracto de código puede valer en ocasiones más que mil palabras, la explicación se realiza a través de ejemplos comentados. Existen diversas formas de procesar este tipo de ficheros; sin embargo, una forma muy sencilla para el programador de escribir en un fichero de texto es mediante la clase `PrintWriter`. Para lectura, es posible utilizar la clase `Scanner`.

15.2.1 Escritura en un fichero de texto

El proceso de escritura en un fichero de texto puede realizarse mediante la clase `PrintWriter` del paquete `java.io`. Esta clase permite gestionar la escritura en un fichero de una manera muy similar a como se muestran los datos por la salida estándar, mediante los métodos `print`, `println` y `printf`; estando los dos primeros métodos sobrecargados para los diferentes tipos de datos primitivos y `String`.

La figura 15.3 muestra un ejemplo de uso de la clase `PrintWriter` para escribir un fichero de texto. Para ello, en primer lugar se importan las clases necesarias del paquete `java.io` (líneas 1-3). Posteriormente, se construye un nuevo objeto de tipo `PrintWriter` invocando a su constructor con un `File` que referencie el fichero de salida (línea 17). En este caso concreto, como se ha especificado una ruta relativa, el fichero se creará en el directorio de trabajo (aquél desde el que se ejecute el programa) si no existía previamente. Si el fichero ya existía previamente, entonces se borra su contenido. A partir de este momento, el objeto `pw` permitirá realizar escrituras en el fichero *file2.txt*.

El constructor de la clase `PrintWriter` puede lanzar la excepción `FileNotFoundException` si el objeto `File` especificado no referencia a un fichero regular que pueda ser escrito o no puede crearse un nuevo fichero. Por ello, se utiliza un bloque `try-catch` (líneas 16-24) para gestionar de forma apropiada dicha excepción.

La clase `PrintWriter` contiene los métodos `print` y `println` para escribir objetos de tipo `String` y cualquier tipo primitivo en Java en el fichero. Su comportamiento es análogo a los correspondientes métodos que se utilizan habitualmente en `System.out` para mostrar por pantalla salvo que, en este caso, escriben los resultados en el fichero (líneas 18-20). En el ejemplo, se están escribiendo dos `String` y un literal de tipo `double`.

Una vez finalizadas las operaciones de escritura, es necesario cerrar el fichero (línea 21) para asegurarse de que los datos realmente se han escrito y liberar los recursos asociados a la gestión del fichero. También puede hacerse en la cláusula `finally` del `try`, comprobando si se ha creado el `PrintWriter` y, si es así, cerrándolo (como

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  /**
5   * Clase TestPrintWriter: uso de la clase PrintWriter para
6   * escribir en un fichero de texto.
7   * @author Libro IIP-PRG
8   * @version 2016
9   */
10 public class TestPrintWriter {
11     /** Método principal.
12      *  @param args String[], argumentos del programa.
13      */
14     public static void main(String[] args) {
15         String fichero = "file2.txt";
16         try {
17             PrintWriter pw = new PrintWriter(new File(fichero));
18             pw.print("El veloz murciélago hindú");
19             pw.println(" comía feliz cardillo y kiwi");
20             pw.println(4.815162342);
21             pw.close();
22         } catch (FileNotFoundException e) {
23             System.err.println("Problemas al abrir el fichero");
24         }
25     }
26 }
```

Figura 15.3: Ejemplo de uso de `PrintWriter` para la escritura en ficheros de texto.

en la figura 15.9). En realidad, si el programador olvida cerrar un fichero, Java se encarga de hacerlo. No obstante, resulta una buena práctica de programación cerrar el fichero una vez finalizado el proceso de escritura (y de lectura). Esto es especialmente necesario dado que los mecanismos de escritura en Java usan generalmente un *buffer*, o almacenamiento temporal intermedio, que permite agrupar los datos de varias escrituras en esta zona temporal hasta que hayan suficientes datos antes de escribirlos en el disco. Este proceso se realiza con el objetivo de hacer más eficiente el proceso de escritura, ya que el acceso al disco físico resulta bastante lento. Bajo este esquema, si no se cierra el fichero y el programa aborta por alguna razón, el fichero puede quedar incompleto.

Volviendo al ejemplo, el contenido del fichero, que puede ser visualizado desde cualquier editor de textos, es el siguiente:

file2.txt

```
El veloz murciélago hindú comía feliz cardillo y kiwi
4.815162342
```

Si ya se dispone de un fichero de texto y se desea añadir nuevos datos al final, hay que modificar la forma de abrirlo, como se muestra a continuación:

```
PrintWriter pw = new PrintWriter(new FileOutputStream(fichero, true));
```

La tabla 15.2 incluye un resumen de los principales constructores y métodos disponibles en la clase `PrintWriter`. Los métodos de dicha clase no lanzan excepciones de E/S. El usuario puede comprobar si ha ocurrido algún error tras una operación de escritura invocando al método `checkError()`.

`public` **`PrintWriter`**`(File f)`	Crea un nuevo `PrintWriter` a partir de un objeto de tipo `File`. Lanza la excepción `FileNotFoundException` si no se puede escribir en el fichero.
`public` **`PrintWriter`**`(OutputStream out)`	Crea un nuevo `PrintWriter` a partir de un flujo de salida.
`public void` **`println`**`(`*arg*`)`	Escribe un determinado argumento *arg* en el stream de salida y termina la línea. El argumento puede ser de cualquier tipo primitivo o un `String`.
`public void` **`print`**`(`*arg*`)`	Mismo funcionamiento que `println` pero no termina la línea.
`public PrintWriter` **`printf`**`(String sf,` *args*`)`	Escribe los argumentos *args* siguiendo la descripción de formato en `sf`.
`public boolean` **`checkError`**`()`	Escribe todos los datos pendientes en el fichero. Si se ha producido algún error durante la escritura devuelve `true`.
`public void` **`close`**`()`	Cierra el `PrintWriter`, liberando recursos.

Tabla 15.2: Clase `PrintWriter`: principales constructores y métodos.

15.2.2 Lectura de un fichero de texto

La lectura de un fichero de texto puede llevarse a cabo utilizando la clase `Scanner` del paquete `java.util`. En capítulos anteriores se utilizó esta clase para poder leer valores desde la entrada estándar. El mecanismo para leer valores de un fichero de texto resulta análogo. Básicamente, esta clase permite abstraer los datos del fichero como si fueran una secuencia de elementos, donde cada uno de ellos puede ser de un tipo diferente. Así, es posible ir leyendo del fichero de texto los elementos

(tipos primitivos, `String` y líneas completas) de forma secuencial, uno a uno. Cada vez que se realiza una operación de lectura de un elemento, el objeto `Scanner` pasa de forma implícita al siguiente elemento o token de la secuencia, que será leído en la siguiente operación de lectura.

La figura 15.4 muestra un ejemplo de utilización de la clase `Scanner` para leer de un fichero de texto llamado *cosas.txt* cuyo contenido se muestra a continuación:

cosas.txt

```
1 2
3 4
Multiplícate por cero!
```

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4  /**
5   * Clase TestScanner: uso de la clase Scanner para
6   * leer de un fichero de texto llamado cosas.txt.
7   * @author Libro IIP-PRG
8   * @version 2016
9   */
10 public class TestScanner {
11     /** Método principal.
12      *  @param args String[], argumentos del programa.
13      */
14     public static void main(String[] args) {
15         System.out.println("Leemos 3 números y una línea de texto");
16         try {
17             Scanner scanner = new Scanner(new File("cosas.txt"));
18             int n1 = scanner.nextInt();
19             int n2 = scanner.nextInt();
20             int n3 = scanner.nextInt();
21             scanner.nextLine();
22             String linea = scanner.nextLine();
23             System.out.println("Números: " + n1 + "," + n2 + "," + n3);
24             System.out.println("La línea es: " + linea);
25             scanner.close();
26         } catch (FileNotFoundException ex) {
27             System.err.println("El fichero no existe." + ex);
28         }
29     }
30 }
```

Figura 15.4: Ejemplo de uso de `Scanner` para leer de un fichero de texto.

En ella se puede observar la creación de un objeto `Scanner` para poder leer del fichero de texto (línea 17). La invocación del constructor de la clase `Scanner` puede lanzar la excepción `FileNotFoundException` si el fichero especificado no existe. Por tanto, es necesario introducir un bloque `try-catch` (líneas 16-28) para especificar las acciones a ejecutar si se produce la excepción.

A continuación se leen tres números enteros utilizando para ello `nextInt()` (líneas 18-20). La invocación al método `nextLine()` de la línea 21 provoca leer el resto de la línea y descartar los datos leídos (el número 4 y el retorno de carro de la línea). La instrucción de la línea 22 permite leer una línea completa. Luego, el programa muestra el resultado de las lecturas por la salida estándar (líneas 23 y 24). Finalmente, el programa cierra el fichero (línea 25).

Nótese que el programa está asumiendo la estructura del fichero y la disposición de sus datos antes de realizar las correspondientes operaciones de lectura. Si se invoca al método `nextInt()` y el valor que existe en el fichero no es un `int`, entonces dicho método lanza la excepción `InputMismatchException`. Como esta excepción es subclase de `RuntimeException` no es necesario gestionarla de forma explícita. Por eso no resulta necesario capturar esa excepción en el código, como ya se ha explicado en el capítulo 14. Más adelante (en la figura 15.7) se verá un ejemplo de lectura de fichero con `nextInt()` en el que se captura la excepción `InputMismatchException`.

A continuación se muestra el contenido de la salida estándar generada por el programa:

Salida Estándar

```
Leemos 3 números y una línea de texto
Números: 1,2,3
La línea es: Multiplícate por cero!
```

La figura 15.5 muestra otro ejemplo de utilización de la clase `Scanner` para leer de un fichero de texto llamado *carreras.txt*. Este fichero incluye el nombre de un corredor y la posición en la que ha finalizado una supuesta carrera, con respecto al resto de corredores. A continuación se muestra un extracto del mismo:

carreras.txt

```
Lucía 7
Enrique 4
María 3
```

En ella se puede observar la creación de un objeto `Scanner` para poder leer del fichero de texto (línea 17). Se observa el uso de un bucle cuya guarda utiliza el método `hasNextLine()` del objeto `Scanner` (línea 18). Esto permite el progreso del bucle hasta que no queden más líneas en el fichero de texto por leer. A continuación

se lee cada una de las líneas mediante el método `nextLine()` (línea 19). Cada línea se divide mediante el método `split(String)` de la clase `String`, usando como separador el espacio. Esto permite obtener todos los tokens (las palabras individuales) de la línea en un array (línea 20). Finalmente, se muestran los valores obtenidos (nombre del corredor y posición) por la salida estándar (línea 21). En el `catch` se captura la excepción `FileNotFoundException` que se puede producir al crear el `Scanner` si el fichero no existe. Y en el `finally`, se comprueba si se ha podido crear y, si es así, se cierra.

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4  /**
5   * Clase TestScannerWhile: uso de la clase Scanner para
6   * leer de un fichero de texto llamado carreras.txt.
7   * @author Libro IIP-PRG
8   * @version 2016
9   */
10 public class TestScannerWhile {
11     /** Método principal.
12      *  @param args String[], argumentos del programa.
13      */
14     public static void main(String[] args) {
15         Scanner sc = null;
16         try {
17             sc = new Scanner(new File("carreras.txt"));
18             while (sc.hasNextLine()) {
19                 String linea = sc.nextLine();
20                 String[] tokens = linea.split(" ");
21                 System.out.println(tokens[0] + " : " + tokens[1]);
22             }
23         } catch (FileNotFoundException ex) {
24             System.err.println("El fichero no existe." + ex);
25         } finally {
26             if (sc != null) { sc.close(); }
27         }
28     }
29 }
```

Figura 15.5: Ejemplo de uso de `Scanner` para leer de un fichero de texto con detección de terminación de fichero.

La tabla 15.3 incluye un resumen de los principales constructores y métodos disponibles en la clase `Scanner`. Se muestran también algunas de las principales excepciones que pueden lanzar dichos métodos.

`public` **Scanner**`(File f)`	Crea un nuevo `Scanner` a partir de un objeto de tipo `File`. Lanza `FileNotFoundException` si el fichero no es accesible.
`public` **Scanner**`(InputStream source)`	Crea un nuevo `Scanner` a partir de un flujo de entrada (véase sección 15.4).
`public String` **next**`()`	Obtiene el siguiente elemento leído como un `String`. Lanza `NoSuchElementException` si no quedan más elementos por leer. Lanza `IllegalStateException` si el flujo del `Scanner` estaba cerrado.
`public String` **nextLine**`()`	Se lee el resto de línea completa, descartando el salto de línea. Devuelve el resultado como un `String`. Lanza `NoSuchElementException` si no quedan más elementos por leer. Lanza `IllegalStateException` si el flujo del `Scanner` estaba cerrado.

`public int` **nextInt**`()`
`public long` **nextLong**`()`
`public short` **nextShort**`()`
`public byte` **nextByte**`()`
`public float` **nextFloat**`()`
`public double` **nextDouble**`()`
`public boolean` **nextBoolean**`()`
	Devuelve el siguiente elemento como un `int` siempre que se trate de un `int`. Ídem para `long`, `short`, `byte`, `float`, `double` y `boolean`. Lanza `InputMismatchException` en caso de no poder obtener un valor del tipo apropiado. Lanza `NoSuchElementException` si no quedan más elementos por leer. Lanza `IllegalStateException` si el flujo del `Scanner` estaba cerrado.

`public boolean` **hasNextInt**`()`
`public boolean` **hasNextLong**`()`
`public boolean` **hasNextShort**`()`
`public boolean` **hasNextByte**`()`
`public boolean` **hasNextFloat**`()`
`public boolean` **hasNextDouble**`()`
`public boolean` **hasNextBoolean**`()`
`public boolean` **hasNext**`()`
`public boolean` **hasNextLine**`()`
	Devuelve `true` si el siguiente elemento a obtener se puede interpretar como un `int`. Ídem para `long`, `short`, `byte`, `float`, `double` y `boolean`. También para detectar si existe un `String` o una línea de texto. Lanza `IllegalStateException` si el flujo del `Scanner` estaba cerrado.

`public Scanner` **useLocale**`(Locale l)`	Establece la configuración local del `Scanner` a la configuración especificada por el `Locale l`.
`public Scanner` **useDelimiter**`(String p)`	Establece el conjunto de delimitadores del `Scanner` a un patrón construido a partir del `String p`.
`public void` **close**`()`	Cierra el `Scanner`, liberando recursos.

Tabla 15.3: Clase `Scanner`: principales constructores y métodos.

Ejemplo 15.1. Recuérdese que la clase `SecuenciaDeCirculos`, definida en la figura 9.18, permite definir una secuencia de círculos mediante un array de objetos `Circulo` (`elArray`) y un entero (`talla`) que indica el número de círculos de que consta la secuencia en cada momento. Supóngase que importa la clase `Circulo` del paquete `figuras` visto en el capítulo 13. Esto es, ahora un círculo de la secuencia tendrá un radio de tipo `double`, un color de tipo `Color` y la posición de su centro será de tipo `Point2D.Double`.

Se quiere almacenar la información de los círculos de la secuencia en un fichero de texto con tantas líneas como círculos tenga la secuencia y cada línea con los datos de un círculo (radio, color, abscisa y ordenada del centro) separados por espacios en blanco. Por ejemplo, los datos de un círculo de radio 50.5, de color verde y cuyo centro está en la posición (100.75, 250.2) se almacenarán en una línea del fichero como: `50.50 0 255 0 100.75 250.20` donde se puede observar que el color se guarda como la combinación de los tres colores básicos rojo, verde y azul (*rgb*). Además, dado un fichero de texto con dicho formato, se desea cargar los círculos en él almacenados en el array de la secuencia. Para ello, en dicha clase, se definen los métodos `guardarFormatoTexto(PrintWriter)` y `cargarFormatoTexto(Scanner)` que se muestran en las figuras 15.6 y 15.7, respectivamente.

Para escribir en un fichero de texto la información de todos los círculos de la secuencia en el formato indicado, en el método `guardarFormatoTexto(PrintWriter)`, dado un `PrintWriter` que representa el fichero donde escribir, se realiza un recorrido de `elArray[0..talla - 1]` y para cada círculo se construye un `String`, con sus datos separados por espacios, que se escribe en el fichero.

```
/** Guarda una SecuenciaDeCirculos en un fichero de texto.
 *  @param pw PrintWriter, para realizar la escritura en fichero.
 */
public void guardarFormatoTexto(PrintWriter pw) {
    for (int i = 0; i < talla; i++) {
        String aux = String.format("%.2f %d %d %d %.2f %.2f",
            elArray[i].getRadio(),
            elArray[i].getColor().getRed(),
            elArray[i].getColor().getGreen(),
            elArray[i].getColor().getBlue(),
            elArray[i].getPosicion().getX(),
            elArray[i].getPosicion().getY());
        pw.println(aux);
    }
}
```

Figura 15.6: Método de escritura en fichero de texto en la clase `SecuenciaDeCirculos`.

Para leer de un fichero de texto que contiene círculos en el formato indicado y cargarlos en `elArray`, en el método `cargarFormatoTexto(Scanner)`, mientras queden tokens por leer del `Scanner` que representa el fichero, en cada pasada del bucle se leen los datos de tipo `double` e `int` de una línea del fichero. Con estos datos, se crea un objeto de tipo `Circulo` que se añade a la secuencia (`this`) usando el método `insertar(Circulo)`. Si en la lectura de algún dato se da un error de formato por no ser un dato válido, los métodos `nextDouble()` y `nextInt()` lanzan la excepción `InputMismatchException` que se captura en el bloque `try-catch`. Además, para garantizar que, tanto si se produce una excepción como si no se produce, la lectura del fichero se completa, se usa una cláusula `finally` con la instrucción `sc.nextLine();` que, en cualquier situación, hace que el `Scanner` avance hacia la siguiente línea del fichero, si la hay.

```
/** Carga los círculos de una SecuenciaDeCirculos desde
 *  un fichero de texto, realizando la lectura con los
 *  métodos nextDouble() y nextInt().
 *  @param sc Scanner, para realizar la lectura de fichero.
 */
public void cargarFormatoTexto(Scanner sc) {
    while (sc.hasNext()) {
        try {
            double radio = sc.nextDouble();
            int red = sc.nextInt();
            int green = sc.nextInt();
            int blue = sc.nextInt();
            double posX = sc.nextDouble();
            double posY = sc.nextDouble();
            Color col = new Color(red, green, blue);
            Circulo circ = new Circulo(radio, col, posX, posY);
            this.insertar(circ);
        } catch (InputMismatchException e) {
            System.out.println("Error de formato en la lectura");
        } finally {
            sc.nextLine();
        }
    }
}
```

Figura 15.7: Método de lectura desde fichero de texto en la clase `SecuenciaDeCirculos`, usando `nextDouble()` y `nextInt()`.

El método `cargarFormatoTexto2(Scanner)` (en la figura 15.8) es una segunda aproximación para realizar la lectura de los círculos del fichero de texto y cargarlos en `elArray`. En este caso, mientras queden datos por leer, en cada pasada del bucle se lee una línea completa del fichero (usando `nextLine()`) que se divide con el método `split(String)` de la clase `String` en los fragmentos apropiados. Usan-

do los métodos `Double.parseDouble(String)` e `Integer.parseInt(String)`, los `String` obtenidos se convierten en los valores numéricos correspondientes a los datos del círculo de la línea leída. Con estos datos, igual que en la primera aproximación, se crea un objeto de tipo `Circulo` que se añade a la secuencia (`this`) usando el método `insertar(Circulo)`. Si alguno de los `String` no se puede convertir en valor numérico, los métodos `parseDouble(String)` y `parseInt(String)` lanzan la excepción `NumberFormatException` que se captura en el bloque `try-catch`, garantizándose la lectura completa del fichero.

```
/** Carga los círculos de una SecuenciaDeCirculos desde
 *  un fichero de texto, realizando la lectura con el método
 *  nextLine() y utilizando el método split(String) de String.
 *  @param sc Scanner, para realizar la lectura de fichero.
 */
public void cargarFormatoTexto2(Scanner sc) {
    while (sc.hasNext()) {
        try {
            String linea = sc.nextLine();
            String[] token = linea.split(" ");
            double radio = Double.parseDouble(token[0]);
            int red = Integer.parseInt(token[1]);
            int green = Integer.parseInt(token[2]);
            int blue = Integer.parseInt(token[3]);
            double posX = Double.parseDouble(token[4]);
            double posY = Double.parseDouble(token[5]);
            Color col = new Color(red, green, blue);
            Circulo circ = new Circulo(radio, col, posX, posY);
            this.insertar(circ);
        } catch (NumberFormatException e) {
            System.out.println("Error de formato en la lectura");
        }
    }
}
```

Figura 15.8: Método de lectura desde fichero de texto en la clase `SecuenciaDeCirculos`, usando `nextLine()` y `split(String)` de `String`.

Desde la clase `GestorSecCirculos`, una clase programa que prueba la clase `SecuenciaDeCirculos`, se realizará la escritura/lectura en/desde fichero de texto con los métodos `guardarSecCircTxt(SecuenciaDeCirculos, String)` y `cargarSecCircTxt(SecuenciaDeCirculos, String)`, respectivamente, que se muestran en las figuras 15.9 y 15.10.

El método `guardarSecCircTxt(SecuenciaDeCirculos, String)`, dada una secuencia de círculos y el nombre de un fichero, crea un objeto `PrintWriter` a partir del `File` del fichero dado y, si se ha creado con éxito, invoca al mètodo

`guardarFormatoTexto(PrintWriter)` de la clase `SecuenciaDeCirculos`, aplicado sobre la secuencia dada, para guardar en el fichero los datos de los círculos de la misma. Si al intentar crear el `PrintWriter` se produce la excepción `FileNotFoundException` (por ejemplo, por falta de espacio en disco o por la no posesión de los permisos adecuados), se captura y se informa al usuario del error ocurrido. Por último, en la cláusula `finally` del `try` se comprueba si se ha creado el `PrintWriter` y, en caso afirmativo, se cierra.

```
/** Guarda una SecuenciaDeCirculos en un fichero de texto.
 *  @param s SecuenciaDeCirculos, cuyos círculos se guardan.
 *  @param nomF String, nombre del fichero donde escribir.
 */
private static void guardarSecCircTxt(SecuenciaDeCirculos s,
    String nomF) {
    PrintWriter fSal = null;
    try {
        fSal = new PrintWriter(new File(nomF));
        s.guardarFormatoTexto(fSal);
    } catch (FileNotFoundException e) {
        String msg = "No se puede escribir en el fichero " + nomF;
        System.out.println(msg);
    } finally {
        if (fSal != null) { fSal.close(); }
    }
}
```

Figura 15.9: Método de escritura en fichero de texto en la clase `GestorSecCirculos`.

El método `cargarSecCircTxt(SecuenciaDeCirculos, String)`, dada una secuencia de círculos y el nombre de un fichero con datos de círculos en el formato anteriormente indicado, crea un objeto de tipo `Scanner` a partir del `File` del fichero dado para realizar la lectura (este objeto se instancia especificando `Locale.US` para que la lectura de los números reales sea correcta, dado el formato de los mismos en el fichero). Al crear este objeto `Scanner`, se intenta localizar un fichero en el directorio en el que se ejecuta la aplicación. Esto puede producir una excepción de tipo `FileNotFoundException` (entre otras razones, porque no existe el fichero o no se poseen los permisos de acceso apropiados) que se captura, informando al usuario del error ocurrido. Si el fichero se ha localizado con éxito, se invoca al método `cargarFormatoTexto(Scanner)` aplicándolo sobre la secuencia de círculos dada (de manera análoga, se podría utilizar el método `cargarFormatoTexto2(Scanner)`). Para finalizar, en la cláusula `finally` del `try` se comprueba si se ha creado el `PrintWriter` y, si es así, se cierra.

```
/** Carga una SecuenciaDeCirculos desde un fichero de texto.
 *  @param s SecuenciaDeCirculos, cuyos círculos se leen.
 *  @param nomF String, nombre del fichero del que leer.
 */
private static void cargarSecCircTxt(SecuenciaDeCirculos s,
    String nomF) {
    Scanner fEnt = null;
    try {
        fEnt = new Scanner(new File(nomF)).useLocale(Locale.US);
        s.cargarFormatoTexto(fEnt);
    } catch (FileNotFoundException e) {
        String msg = "No se puede acceder al fichero " + nomF;
        System.out.println(msg);
    } finally {
        if (fEnt != null) { fEnt.close(); }
    }
}
```

Figura 15.10: Métodos de lectura desde fichero de texto en la clase `GestorSecCirculos`.

15.3 Ficheros binarios

Esta sección aborda el proceso de escritura y lectura de ficheros binarios, a través de ejemplos comentados. La forma más cómoda para el programador de escribir en un fichero binario es mediante la clase `ObjectOutputStream`. Para lectura, es posible utilizar la clase `ObjectInputStream`[3].

15.3.1 Escritura en un fichero binario

La escritura se realizará mediante la clase `ObjectOutputStream`, del paquete `java.io`, que permite la escritura de tipos primitivos y `String` en un fichero binario. En la sección 15.5 se abordará la escritura y la lectura de objetos en/desde este tipo de ficheros.

La figura 15.11 muestra un ejemplo de escritura en un fichero binario de tres valores de diferentes tipos (un `String`, un `int` y un `double`). Por lo tanto, un fichero binario no tiene porqué ser homogéneo sino que puede almacenar diferentes tipos de datos. En primer lugar, se procede a la construcción del objeto `ObjectOutputStream` (líneas 21 y 22). A continuación, se utilizan los métodos correspondientes para guardar los diferentes valores, dependiendo del tipo de da-

[3]En realidad también se podrían utilizar las clases `DataOutputStream` y `DataInputStream`, con idéntica semántica y funcionalidad salvo que, estas últimas, no permiten la escritura de objetos completos en ficheros, tal y como se verá en la sección 15.5.

tos: `writeUTF(String)`, para escribir un `String`; `writeInt(int)` para escribir un entero y `writeDouble(double)` para escribir un valor decimal de doble precisión.

```
1  import java.io.FileInputStream;
2  import java.io.FileOutputStream;
3  import java.io.IOException;
4  import java.io.ObjectInputStream;
5  import java.io.ObjectOutputStream;
6  /**
7   * Clase Calificaciones: escritura de un String, un int y un
8   * double en un fichero binario y posterior lectura del mismo.
9   * @author Libro IIP-PRG
10  * @version 2016
11  */
12 public class Calificaciones {
13     /** Método principal.
14      *  @param args String[], argumentos del programa.
15      */
16     public static void main(String[] args) {
17         String fichero = "calificaciones.dat";
18         String nombre = "IIP";
19         int conv = 1; double nota = 7.8;
20         try {
21             ObjectOutputStream out =
22                 new ObjectOutputStream(new FileOutputStream(fichero));
23             out.writeUTF(nombre);
24             out.writeInt(conv);
25             out.writeDouble(nota);
26             out.close();
27             ObjectInputStream in =
28                 new ObjectInputStream(new FileInputStream(fichero));
29             System.out.println("Nombre leído: " + in.readUTF());
30             System.out.println("Convocatoria leída: " + in.readInt());
31             System.out.println("Nota leída: " + in.readDouble());
32             in.close();
33         } catch (IOException e) {
34             System.err.println("Problemas con el fichero.");
35         }
36     }
37 }
```

Figura 15.11: Ejemplo de escritura y posterior lectura de un fichero binario.

Finalizadas las escrituras, resulta conveniente cerrar el fichero para liberar los recursos asociados y garantizar la escritura de los datos en disco (línea 26). También puede hacerse, como se ha visto para los ficheros de texto, en la cláusula `finally` del `try`, comprobando si se ha creado el `ObjectOutputStream` y, si es así, ce-

rrándolo. Pero como el método `close()` de `ObjectOutputStream` puede lanzar la excepción `IOException`, en este caso, habrá que capturar dicha excepción (como en la figura 15.26).

El fichero generado en disco, denominado *calificaciones.dat*, contiene la secuencia de bytes que representan los valores almacenados. Por lo tanto, no se trata de un fichero visualizable ni editable directamente. Debe ser leído desde código Java.

Las líneas 27-32 del ejemplo se centran en la posterior lectura de los datos escritos en el fichero para verificar su correcta funcionalidad. Nótese el paralelismo entre la nomenclatura de los métodos utilizados para escribir en un fichero y los usados para leer del mismo. No obstante, en la siguiente sección se aborda y describe de forma individualizada el proceso de lectura de un fichero binario.

El contenido de la salida estándar generada por el programa se muestra a continuación:

Salida Estándar

```
Nombre leído: IIP
Convocatoria leída: 1
Nota leída: 7.8
```

La tabla 15.4 incluye un resumen de los principales constructores y métodos disponibles en la clase `ObjectOutputStream` cuando la misma se utiliza para escribir en ficheros binarios datos elementales y de tipo `String` en formato UTF-8. Más adelante se ampliará esta tabla con las operaciones para poder escribir objetos en ese tipo de ficheros.

15.3.2 Lectura de un fichero binario

Para ejemplificar la lectura de un fichero binario, la figura 15.12 muestra un código que permite leer de un fichero binario que incluye un conjunto de medidas de precipitación para una ciudad concreta a lo largo de un mes. Como no todos los días llueve, tan solo se incluyen datos para los días que ha llovido. Cada dato es el número de litros por metro cuadrado registrados en dicha ciudad para ese día concreto. Por lo tanto, para cada ciudad hay un número variable de datos. El fichero tiene la siguiente estructura:

lluvias.dat

```
Nombre_Ciudad N Dato_0 Dato_1 ... Dato_N-1
```

Se observa que en primer lugar aparece el nombre de la ciudad, a continuación el número de datos de precipitación (`N`) y, luego, los `N` datos de precipitación. Recuérdese que, al tratarse de un fichero binario, no es posible visualizar los datos del mismo con un editor de ficheros, sino que debe leerse desde un programa Java.

`public` **ObjectOutputStream**`(OutputStream o)`	Crea un `ObjectOutputStream` a partir del flujo de salida `o`. Lanza `IOException` si no se puede escribir en el flujo.

`public void` **writeInt**`(int v)`
`public void` **writeLong**`(long v)`
`public void` **writeShort**`(short v)`
`public void` **writeByte**`(byte v)`
`public void` **writeFloat**`(float v)`
`public void` **writeDouble**`(double v)`
`public void` **writeBoolean**`(boolean v)`
`public void` **writeChar**`(char v)`
	Escribe el valor elemental `v`, del tipo nombrado por el método (`int`, `long`, `short`, `double`, etc.), en binario, en el flujo de salida asociado. Lanza `IOException` si no se puede escribir en el flujo.

`public void` **writeUTF**`(String str)`	Escribe el `String str`, codificado en UTF-8 modificado, en el flujo de salida asociado. Lanza `IOException` si no se puede escribir en el flujo.
`public void` **writeChars**`(String str)`	Escribe el `String str`, carácter a carácter, en el flujo de salida asociado. Lanza `IOException` si no se puede escribir en el flujo.
`public void` **close**`()`	Cierra el flujo de salida, liberando recursos. Lanza `IOException` si no se puede cerrar.

Tabla 15.4: Clase `ObjectOutputStream`: principales constructores y métodos para escribir valores elementales y de tipo `String` en flujos de datos binarios.

En el ejemplo, en primer lugar se construye el objeto de tipo `ObjectInputStream` que permite leer los datos del fichero binario (líneas 18 y 19). A continuación, se lee el nombre de la ciudad y el número de datos de precipitación que vendrán a continuación (líneas 20 y 21). En ese preciso momento ya es posible declarar y construir el array `lluvias` (línea 22) para ajustarlo al número de datos que se leerán. En este sentido, comienza un bucle de lectura de los valores de precipitación (líneas 23-25) que permite leer los datos del fichero y almacenarlos en el array. Finalizado el proceso de lectura, se procede a cerrar el fichero (línea 26). Por último, se muestran algunos datos leídos por la salida estándar (líneas 27-29).

La invocación del constructor de `FileInputStream` puede lanzar la excepción `FileNotFoundException` si ocurre algún problema tratando de abrir el fichero. Además, los métodos de lectura de tipos pueden lanzar la excepción `IOException` en caso de problemas al realizar la operación. Como `FileNotFoundException` es subclase de `IOException`, es posible realizar la gestión de excepciones que aparece

```
1  import java.io.FileInputStream;
2  import java.io.IOException;
3  import java.io.ObjectInputStream;
4  /**
5   * Clase TestObjectInputStream: lectura de un fichero binario
6   * con un conjunto de medidas de precipitación para una ciudad
7   * concreta a lo largo de un mes.
8   * @author Libro IIP-PRG
9   * @version 2016
10  */
11 public class TestObjectInputStream {
12     /** Método principal.
13      *  @param args String[], argumentos del programa.
14      */
15     public static void main(String[] args) {
16         String fichero = "lluvias.dat";
17         try {
18             ObjectInputStream in =
19                 new ObjectInputStream(new FileInputStream(fichero));
20             String ciudad = in.readUTF();
21             int nDatos = in.readInt();
22             float[] lluvias = new float[nDatos];
23             for (int i = 0; i < nDatos; i++) {
24                 lluvias[i] = in.readFloat();
25             }
26             in.close();
27             System.out.println("Ciudad: " + ciudad);
28             System.out.println("Nº de datos: " + nDatos);
29             System.out.println("Primer dato: " + lluvias[0]);
30         } catch (IOException ex) {
31             System.err.println("Problemas al leer: " + ex);
32         }
33     }
34 }
```

Figura 15.12: Ejemplo de lectura de un fichero binario.

en las líneas 17-32. No obstante, el programador podría decidir realizar un tratamiento diferenciado de ambas excepciones para distinguir entre el error provocado por no encontrar un fichero y el resultante de un error al realizar una operación de E/S, por ejemplo, derivado de un fallo en el sistema de almacenamiento.

El cierre del fichero (línea 26) también podría hacerse en la clásula `finally` del `try`, comprobando si el `ObjectInputStream` se ha podido crear y, si es así, cerrándolo. Pero como el método `close()` de `ObjectInputStream` puede lanzar la excepción

`IOException`, las instrucciones de comprobación y cierre deberían incluirse en un bloque `try-catch` que capture dicha excepción (como en la figura 15.27).

La tabla 15.5 incluye un resumen de los principales constructores y métodos disponibles en la clase `ObjectInputStream` cuando se utiliza para leer de ficheros binarios datos elementales y de tipo `String` en formato UTF-8. Más adelante se ampliará esta tabla con las operaciones para poder leer objetos de ficheros de este tipo.

`public` **`ObjectInputStream`**`(InputStream i)`	Crea un `ObjectInputStream` a partir del flujo de entrada `i`. Lanza `IOException` si no se puede leer del flujo.
`public int` **`read`**`()`	Lee un byte de datos y devuelve dicho byte o -1 si no hay entrada disponible. Lanza `IOException` si se produce un error de E/S.

`public int` **`readInt`**`()`
`public long` **`readLong`**`()`
`public short` **`readShort`**`()`
`public byte` **`readByte`**`()`
`public float` **`readFloat`**`()`
`public double` **`readDouble`**`()`
`public boolean` **`readBoolean`**`()`
`public char` **`readChar`**`()`
	Lee un valor elemental, del tipo nombrado por el método (`int`, `long`, `short`, etc.), desde el flujo de entrada asociado. Lanza `EOFException` si se llega al final del fichero. Lanza `IOException` si se produce un error de E/S.

`public String` **`readUTF`**`()`	Lee un `String`, codificado en UTF-8 modificado, del flujo de entrada asociado. Lanza `IOException` si se produce un error de E/S.
`public void` **`close`**`()`	Cierra el flujo de entrada, liberando recursos. Lanza `IOException` si se produce un error de E/S.

Tabla 15.5: Clase `ObjectInputStream`: principales constructores y métodos para leer valores elementales y de tipo `String` de flujos de datos binarios.

15.3.3 Ficheros binarios de acceso aleatorio

Las secciones anteriores se han centrado en ficheros de acceso secuencial, donde los datos deben ser leídos en el mismo orden en el que fueron escritos. Sin embargo, existen aplicaciones que pueden beneficiarse de acceder a cualquier punto del fichero sin necesidad de haber leído previamente todos los datos anteriores. Volviendo al ejemplo que se mostró en la figura 15.1, si se conoce el punto exacto

del registro al que se pretende acceder en el fichero es mucho más eficiente leer única y exclusivamente ese registro sin necesidad de tener que procesar todos los registros anteriores. Esto podría ser posible si se conociese el tamaño de cada uno de los campos, lo que permitiría saber el tamaño de cada registro y, por lo tanto, la posición dentro del fichero de cada registro.

Java dispone de la clase `RandomAccessFile` que permite el acceso aleatorio a un fichero binario que representa un conjunto de bytes. Se dispone de un puntero con una granularidad de 1 byte que el programador puede mover por el fichero para realizar operaciones de lectura y escritura en cualquier punto del fichero. Para ello, es necesario que los datos guardados en el fichero tengan un tamaño coherente con su tipo de datos. Por ejemplo, un `int` se almacena con un tamaño de 4 bytes, un `double` requiere 8 bytes y un `boolean` se guarda como 1 byte. En el caso especial de los `String`, se escriben primero dos bytes que indican el número de bytes que vienen a continuación y que representan el `String`. Es importante saber que cada operación de lectura y escritura provoca el avance del puntero del fichero tantos bytes como se hayan leído o escrito. En este sentido, conocer el tamaño exacto de cada campo en el fichero facilita calcular el desplazamiento necesario dentro del fichero para acceder a la posición de un determinado dato.

La figura 15.13 muestra un ejemplo de uso de ficheros de acceso aleatorio en Java. En primer lugar, se procede a la creación del fichero indicando que se va a utilizar tanto para lectura como para escritura (`rw` = read / write) (línea 15). A continuación, se escriben dos valores enteros, que ocupan 4 bytes cada uno en el fichero (líneas 17 y 18) y se consulta la longitud del fichero, que será 8 bytes, así como la posición del puntero, que estará en el 8º byte (líneas 19 y 20). En este momento, se desplaza el puntero al 4º byte (línea 22), que representa el comienzo del almacenamiento del número 89. En efecto, los bytes 0, 1, 2 y 3 se emplean para almacenar el entero 65, por lo que el 4º byte es el comienzo del número 89. En ese momento, se procede a leer un entero, que será el número 89 (línea 23). A continuación se sobrescribe el valor 89 por el 77 (líneas 26 y 27) y se verifica la lectura de ese nuevo entero (líneas 29-31). Finalmente, se lleva el puntero al final del fichero y se escribe un valor en doble precisión (`double`) que requiere 8 bytes de almacenamiento (líneas 33-34). Por ello, para releer el valor no hay más que atrasar el puntero 8 bytes y realizar una operación de lectura de un `double` (líneas 36-37). Por último, se cierra el fichero (línea 39).

Nótese que el constructor de la clase `RandomAccessFile` puede lanzar la excepción `FileNotFoundException` mientras que los métodos pueden lanzar `IOException`. Como la primera es subclase de la segunda, se utiliza un bloque `try-catch` general de gestión de excepciones.

La ejecución del programa muestra el siguiente resultado por la salida estándar:

Salida Estándar

```
Se escriben dos enteros (65 y 89):
Longitud: 16
Puntero: 8
Moviendo el puntero al 4º byte
Entero leído: 89
Machacando el entero por 77
Verificando el valor sobrescrito
Entero leído: 77
Llevando el puntero al final
Releyendo el valor escrito
Valor leído: 178.54
```

Como siempre, se recomienda al lector consultar el *API* de Java [Ora16b] para obtener más información sobre la clase `RandomAccessFile`.

15.4 Otros tipos de flujos

En las secciones anteriores se han mostrado las formas más convenientes para acceder a ficheros tanto de texto como binarios. No obstante, la E/S en Java permite funcionalidades mucho más allá de leer y escribir en ficheros. Como se comentó al inicio del capítulo, la E/S está basada en flujos, que no son más que una secuencia de bytes que parten de un origen y se dirigen a un destino. Los flujos que se originan en el programa se llaman *flujos de salida* mientras que los que sirven como entrada de datos al programa se llaman *flujos de entrada*. En Java existen dos grandes categorías de flujos:

- *Flujos de bytes*. Permiten manejar de forma eficiente la E/S de bytes. Se usan generalmente al leer o escribir datos binarios.
- *Flujos de caracteres*. Permiten gestionar la E/S de caracteres. Se usan al leer o escribir datos de texto. Se utiliza *Unicode* como esquema de codificación, soportando así la diversidad de caracteres de diferentes lenguas.

Cada categoría de flujos supone una jerarquía diferente de clases. Por lo tanto, el número de clases involucradas en la E/S en Java es bastante elevado. Cabe destacar que, en el nivel más bajo, la E/S está orientada a bytes. Los flujos de caracteres tan solo proporcionan una capa de abstracción por encima para poder gestionar los caracteres.

```
1  import java.io.IOException;
2  import java.io.RandomAccessFile;
3  /**
4   * Clase TestRandomAccessFile: lectura y escritura en un fichero
5   * de acceso aleatorio.
6   * @author Libro IIP-PRG
7   * @version 2016
8   */
9  public class TestRandomAccessFile {
10     /** Método principal.
11      *  @param args String[], argumentos del programa.
12      */
13     public static void main(String[] args) {
14         try {
15             RandomAccessFile raf = new RandomAccessFile("data", "rw");
16             System.out.println("Se escriben dos enteros (65 y 89):");
17             raf.writeInt(65);
18             raf.writeInt(89);
19             System.out.println("Longitud: " + raf.length());
20             System.out.println("Puntero: " + raf.getFilePointer());
21             System.out.println("Moviendo el puntero al 4º byte");
22             raf.seek(4);
23             int a = raf.readInt();
24             System.out.println("Entero leído: " + a);
25             System.out.println("Machacando el entero por 77");
26             raf.seek(4);
27             raf.writeInt(77);
28             System.out.println("Verificando el valor sobrescrito");
29             raf.seek(4);
30             int b = raf.readInt();
31             System.out.println("Entero leído: " + b);
32             System.out.println("Llevando el puntero al final");
33             raf.seek(raf.length());
34             raf.writeDouble(178.54);
35             System.out.println("Releyendo el valor escrito");
36             raf.seek(raf.getFilePointer() - 8);
37             double c  = raf.readDouble();
38             System.out.println("Valor leído: " + c);
39             raf.close();
40         } catch (IOException ex) {
41             System.err.println("Problemas durante la E/S" + ex);
42         }
43     }
44 }
```

Figura 15.13: Ejemplo de lectura y escritura en fichero de acceso aleatorio.

15.4.1 Flujos de bytes

Los flujos de bytes están representados por dos clases abstractas `InputStream` y `OutputStream`, encargadas de los flujos de entrada y de salida, respectivamente.

La figura 15.14 muestra gran parte de la jerarquía de clases de `InputStream`, que permite la definición y uso de flujos de entrada al programa. Por ejemplo, la clase `FileInputStream` permite definir un flujo de entrada de bytes para leer desde un fichero binario. No obstante, a partir de ese flujo tan solo es posible leer un conjunto de bytes, no es posible interpretar directamente los valores como los correspondientes tipos que puedan estar representando ese grupo de bytes. Por ello existen determinadas subclases que permiten procesar esa información y manipular los tipos de datos que realmente están representando esos grupos de bytes. Este es el caso de la clase `ObjectInputStream`, utilizada anteriormente, que permite la lectura de datos de distintos tipos a través de un flujo de bytes.

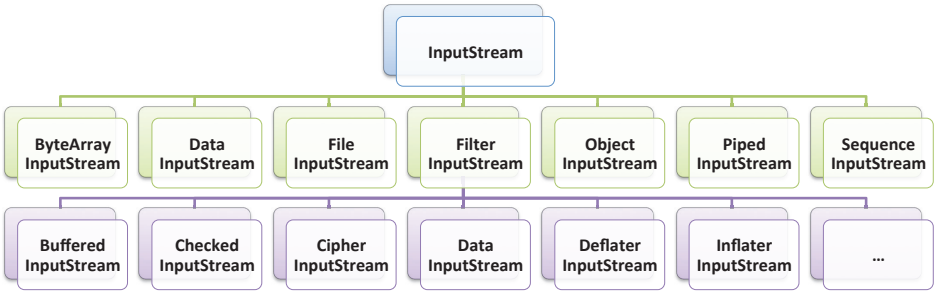

Figura 15.14: Jerarquía de clases a partir de `InputStream` (flujos binarios).

Es posible realizar funcionalidad avanzada mediante estas clases, como la lectura de un flujo de bytes procedentes de una entrada cifrada cuyos datos sean descifrados al mismo tiempo que se leen (`CipherInputStream`). También es posible leer desde un flujo de entrada de bytes procedentes de una fuente de datos comprimida (`DeflaterInputStream`). Se recomienda al lector acudir al *API* de Java [Ora16b] para investigar sobre el funcionamiento del resto de clases. La jerarquía de clases para `OutputStream` es prácticamente idéntica a la mostrada, salvo que todas las clases acaban con el sufijo `OutputStream`.

Para ejemplificar la E/S sobre un elemento que no sea un fichero, la figura 15.15 muestra un ejemplo de lectura a partir de una *URL*. El ejemplo permite leer de una conexión al servidor disponible en `http://www.google.com`. En primer lugar se construye el objeto `URL` (línea 17), y se abre una conexión con el servidor (línea 18). Posteriormente, se obtiene un `InputStream` para poder leer desde la conexión con el servidor. Ese `InputStream` se pasa como argumento a un objeto `Scanner` para poder ir leyendo línea a línea (línea 19). El bucle (líneas 20-23) permite ir recuperando las líneas y mostrarlas por pantalla. Se capturan todas las posibles excepciones de E/S que se puedan producir, capturando la excepción `IOException` y en la cláusula `finally` se cierra el `Scanner`, si se ha podido crear.

```
1  import java.io.IOException;
2  import java.net.URL;
3  import java.net.URLConnection;
4  import java.util.Scanner;
5  /**
6   * Clase TestURL: lectura a partir de una URL.
7   * @author Libro IIP-PRG
8   * @version 2016
9   */
10 public class TestURL {
11     /** Método principal.
12      *  @param args String[], argumentos del programa.
13      */
14     public static void main(String[] args) {
15         Scanner sc = null;
16         try {
17             URL url = new URL("http://www.google.com");
18             URLConnection con = url.openConnection();
19             sc = new Scanner(con.getInputStream());
20             while (sc.hasNextLine()) {
21                 String l = sc.nextLine();
22                 System.out.println(l);
23             }
24         } catch (IOException ex) {
25             System.err.println("Error: " + ex);
26         } finally {
27             if (sc != null) { sc.close(); }
28         }
29     }
30 }
```

Figura 15.15: Ejemplo de lectura a partir de una URL.

A continuación se muestra un extracto del resultado obtenido por la salida estándar. Se trata del código HTML recuperado por la conexión al servidor.

Salida Estándar

```
<!doctype html><html><head><meta http-equiv="content-type"
 content="text/html; charset=ISO-8859-1"><title>Google</title>
...
```

15.4.2 Flujos de caracteres

Los flujos de caracteres en Java están representados por las clases abstractas `Reader` y `Writer`. Ambos operan sobre flujos de caracteres codificados en *Unicode*, soportando así los múltiples caracteres de lenguas internacionales.

La figura 15.16 muestra la jerarquía de clases que heredan de `Writer`. Aquí aparece la clase `PrintWriter` que ha sido utilizada en las secciones anteriores para escritura en ficheros de texto. Esta jerarquía de clases proporciona funcionalidades adicionales. Por ejemplo, la clase `PipedWriter` permite comunicar dos procesos (o dos hilos de ejecución) a través de una tubería para que compartan datos basados en texto entre ellos. Se invita al lector a que consulte el *API* de Java [Ora16b] para conocer la funcionalidad del resto de clases.

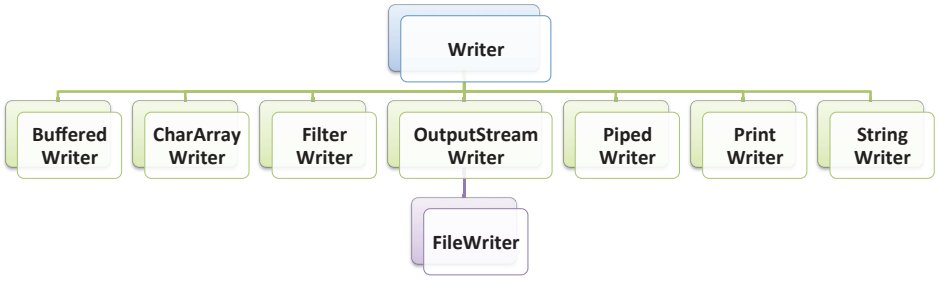

Figura 15.16: Jerarquía de clases a partir de la clase `Writer` (flujos de caracteres).

15.5 E/S de objetos

En las secciones anteriores se ha abordado el proceso de lectura y escritura de tipos primitivos y `String`. No obstante, en un paradigma de orientación a objetos como es el que ofrece Java, los programadores trabajan con objetos que tienen un determinado estado representado por los valores de los atributos. Estos atributos pueden ser tanto tipos primitivos y `String` como referencias a otros objetos.

Java permite almacenar objetos completos (todos sus atributos, incluidas las referencias a otros objetos) en un flujo de salida, que puede ser almacenado en un fichero. Esto permite almacenar grupos de objetos que están en memoria en un archivo. Posteriormente, es posible leer dicho fichero para volver a obtener en memoria todos esos objetos. De esta manera, un grupo de objetos puede sobrevivir a la ejecución de un programa. Todo este proceso, que a priori es complejo, se gestiona en Java de una manera muy sencilla para el programador. El proceso anterior está descrito en la figura 15.17, que menciona las clases necesarias para llevarlo a cabo.

Figura 15.17: E/S de objetos en ficheros binarios.

Las clases cuyos objetos se quieren guardar en el disco deben implementar la interfaz `Serializable`. Se trata de una interfaz que no contiene métodos, por lo que tan solo es una forma de etiquetar a aquellas clases cuyos objetos pueden estar involucrados en operaciones de E/S. Por ejemplo, la figura 15.18 muestra la clase `Persona`, que define tres atributos de diferentes tipos para representar el nombre, la edad y la altura y que implementa la interfaz `Serializable`.

Para poder almacenar los objetos en disco se utiliza la clase estándar `ObjectOutputStream` que permite convertir los objetos a un flujo de bytes que es escrito mediante un `FileOutputStream`.

En la tabla 15.4 se vieron los principales métodos de la clase `ObjectOutputStream` cuando se utiliza para trabajar exclusivamente con valores elementales o de tipo `String`. En la tabla 15.6, complementaria de la anterior, se describe el método `writeObject(Object)`, de `ObjectOutputStream`, mediante el que es posible escribir un objeto en un flujo de salida binario.

`public void` **`writeObject`**`(Object obj)`	Escribe el objeto `obj` en el flujo de salida asociado. Lanza `IOException` si se produce una situación anómala en la E/S.

Tabla 15.6: Método de la clase `ObjectOutputStream` para escribir un objeto en un flujo de datos binarios.

Análogamente, la lectura de objetos se hace a partir de un `FileInputStream` encargado de leer el flujo de bytes que son posteriormente utilizados para la creación del objeto en memoria a partir del `ObjectInputStream`. Al igual que antes, en la tabla 15.5 se vieron los principales métodos de la clase `ObjectInputStream` cuando se utiliza para trabajar con valores elementales o de tipo `String`. En la tabla 15.7, complementaria de la anterior, se describe el método `readObject()`, de `ObjectInputStream`, mediante el que es posible leer un objeto de un flujo de entrada binario.

```java
import java.io.Serializable;
/**
 * Clase Persona: representa una persona por su nombre, edad y altura.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Persona implements Serializable {
    private String nombre;
    private int edad;
    private double altura;

    /** Crea una Persona de nombre n, edad e y altura a.
     *  @param n String, el nombre.
     *  @param e int, la edad.
     *  @param a double, la altura.
     */
    public Persona(String n, int e, double a) {
        nombre = n; edad = e; altura = a;
    }

    /** Devuelve un String con los datos de la Persona.
     *  @return String, representación de los datos.
     */
    public String toString() {
        return nombre + ", " + edad + ", " + altura;
    }
}
```

Figura 15.18: Ejemplo de clase que implementa la interfaz `Serializable`.

`public Object` **`readObject()`**	Lee un objeto desde el flujo de entrada. Lanza `ClassNotFoundException` si no se puede obtener la clase del objeto leído. Lanza `IOException` si se produce una situación anómala en la E/S.

Tabla 15.7: Método de la clase `ObjectInputStream` para leer un objeto de un flujo de datos binarios.

Nótese que el uso de flujos permitiría canalizar la escritura/lectura de objetos hacía un flujo de salida/entrada que conectase con otro proceso, lo que permitiría transferir objetos de un proceso a otro, incluso en diferentes máquinas. Por lo tanto, este proceso no se restringe exclusivamente a ficheros.

La figura 15.19 muestra un ejemplo donde se realiza la E/S de un objeto de tipo `Persona` (aunque podrían haber sido múltiples objetos, incluso de diferentes tipos).

En primer lugar se procede a la creación del flujo de salida de objetos mediante la clase `ObjectOutputStream` (líneas 20 y 21). A continuación, se procede a la escritura del objeto utilizando el método `writeObject(Object)` (línea 22). Finalmente, se procede a cerrar el flujo de salida (línea 23), como es habitual. Después, se procede a la creación del flujo de entrada de objetos mediante la clase `ObjectInputStream` (líneas 24 y 25). A continuación, se procede a la lectura del objeto utilizando el método `readObject()` (línea 26). Nótese el casting al tipo `Persona` del `Object` leído. Por último, se cierra el flujo de entrada (línea 27) y se muestra en la salida estándar la información del objeto `Persona` leído (línea 28).

```
1  import java.io.FileInputStream;
2  import java.io.FileOutputStream;
3  import java.io.IOException;
4  import java.io.ObjectInputStream;
5  import java.io.ObjectOutputStream;
6  /**
7   * Clase TestObjectIO: E/S de un objeto de tipo Persona
8   * contenido en el fichero obj.data.
9   * @author Libro IIP-PRG
10  * @version 2016
11  */
12 public class TestObjectIO {
13     /** Método principal.
14      *  @param args String[], argumentos del programa.
15      */
16     public static void main(String[] args) {
17         String filename = "obj.data";
18         Persona p = new Persona("Luisa Garcia", 25, 179.45);
19         try {
20             ObjectOutputStream ous =
21                 new ObjectOutputStream(new FileOutputStream(filename));
22             ous.writeObject(p);
23             ous.close();
24             ObjectInputStream ois =
25                 new ObjectInputStream(new FileInputStream(filename));
26             Persona p2 = (Persona) ois.readObject();
27             ois.close();
28             System.out.println("Persona leída: " + p2.toString());
29         } catch (IOException ex1) {
30             System.err.println("Error de E/S: " + ex1);
31         } catch (ClassNotFoundException ex2) {
32             System.err.println("Clase no encontrada: " + ex2);
33         }
34     }
35 }
```

Figura 15.19: Ejemplo de uso de E/S de objetos.

Como se puede observar, la lectura/escritura de objetos en ficheros es muy transparente en Java. Además, es posible almacenar clases cuyos atributos sean referencias

a otras clases definidas por el usuario. En este caso, se procesan de forma recursiva todos los objetos, almacenando todos los objetos involucrados. Para ello, todas las clases involucradas deben implementar la interfaz `Serializable`.

Un ejemplo algo más elaborado, en el que también se utiliza la E/S de objetos, puede verse en las figuras 15.20, 15.21 y 15.22 en las que se presenta de forma abreviada las clases `ItemAgenda`, para representar información básica de una entrada individual de una agenda, `Agenda`, que modela una lista de tales entradas básicas (gestionada internamente mediante un array de dimensión ampliable) y la clase `GestorPrueba`, que permite interactuar de forma elemental con las primeras, mediante la creación, almacenamiento y recuperación de una `Agenda` con algunos elementos.

```
1  import java.io.Serializable;
2  /**
3   * Clase ItemAgenda: representa un contacto de una agenda telefónica
4   * que incluye nombre, teléfono y código postal.
5   * @author Libro IIP-PRG
6   * @version 2016
7   */
8  public class ItemAgenda implements Serializable {
9      private String nom, tel;
10     private int postal;
11
12     /** Crea un ItemAgenda de nombre n, teléfono t y código postal p.
13      *  @param n String, el nombre.
14      *  @param t String, el teléfono.
15      *  @param p int, el código postal.
16      */
17     public ItemAgenda(String n, String t, int p) {
18         nom = n; tel = t; postal = p;
19     }
20
21     /** Devuelve un String con los datos del ItemAgenda.
22      *  @return String, representación de los datos.
23      */
24     public String toString() {
25         return nom + ": " + tel + " (" + postal + ")";
26     }
27
28     // Otros métodos de la clase ItemAgenda ...
29 }
```

Figura 15.20: La clase `ItemAgenda` abreviada.

Como se puede obervar, un objeto `Agenda` puede contener varios `ItemAgenda`. Como la política de almacenamiento consiste en guardar por completo un objeto `Agenda`, para poder recuperarlo posteriormente, es necesario explicitar que ambas clases implementan la interfaz `Serializable`.

```
1  import java.io.File;
2  import java.io.FileInputStream;
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.io.Serializable;
8  /**
9   *  Clase Agenda: representa los contactos de una agenda telefónica
10  *  sobre un array, con la funcionalidad que aparece a continuación.
11  *  @author Libro IIP-PRG
12  *  @version 2016
13  */
14 public class Agenda implements Serializable {
15      public static final int MAX = 100;
16      private ItemAgenda[] elArray;
17      private int num;
18
19      /** Crea una agenda vacía (sin contactos). */
20      public Agenda() { elArray = new ItemAgenda[MAX]; num = 0; }
21
22      /** Añade un contacto al final de la agenda.
23       *  Duplica el tamaño del array si es necesario.
24       *  @param b ItemAgenda, el contacto a añadir.
25       */
26      public void insertar(ItemAgenda b) {
27          if (num >= elArray.length) { duplicaEspacio(); }
28          elArray[num++] = b;
29      }
30
31      /**  Duplica el tamaño del atributo elArray de la agenda,
32       *   manteniendo sus elementos.
33       */
34      private void duplicaEspacio() {
35          ItemAgenda[] aux = new ItemAgenda[2 * elArray.length];
36          for (int i = 0; i < elArray.length; i++) { aux[i] = elArray[i]; }
37          elArray = aux;
38      }
39
40      /** Devuelve un String con toda la información de los contactos
41       *  de la agenda.
42       *  @return String, representación de los contactos.
43       */
44      public String toString() {
45          String res = "";
46          for (int i = 0; i < num; i++) { res += elArray[i] + "\n"; }
47          res += "================================";
48          return res;
49      }
50
51      // Otros métodos de la clase Agenda ...
52
```

Figura 15.21: Clase `Agenda`.

```
53      /** Guarda la Agenda en un fichero binario.
54       *  @param fich String, el nombre del fichero.
55       */
56      public void guardarAgenda(String fich) {
57          try {
58              ObjectOutputStream oos = new ObjectOutputStream(
59                  new FileOutputStream(new File(fich)));
60              oos.writeObject(this);
61              oos.close();
62          } catch (IOException fex) {
63              System.err.println("Error al guardar: " + fex.getMessage());
64          }
65      }
66
67      /** Devuelve una Agenda, leída desde un fichero binario.
68       *  @param fich String, el nombre del fichero.
69       */
70      public static Agenda leerAgenda(String fich) {
71          Agenda aux = null;
72          try {
73              ObjectInputStream ois = new ObjectInputStream(
74                  new FileInputStream(fich));
75              aux = (Agenda) ois.readObject();
76              ois.close();
77          } catch (IOException ex) {
78              System.err.println("Error al recuperar: " + ex.getMessage());
79          } catch (ClassNotFoundException ex) {
80              System.err.println("Clase no coincidente:" + ex.getMessage());
81          }
82          return aux;
83      }
84  }
```

Figura 15.21: Clase `Agenda` (cont.).

La gestión del almacenamiento y recuperación se ha definido en la propia clase `Agenda`, cuyos objetos deben ser almacenados y/o recuperados (puede verse en la figura 15.21). Para el almacenamiento se ha definido el método de objeto `guardarAgenda(String)` (línea 56), mientras que para recuperarla se ha definido el método de clase `leerAgenda(String)` (línea 70).

Cuando se ejecuta el `main` de la clase `GestorPrueba`, se construye una `Agenda`, a partir de algunos `ItemAgenda` (líneas 11 a 17, en la figura 15.22) para, a continuación, almacenarla en el fichero de objetos `agenda1.dat` (línea 20) del que se lee y recupera posteriormente (línea 25).

```
1  /**
2   * Clase GestorPrueba: clase programa que prueba la clase Agenda.
3   * @author Libro IIP-PRG
4   * @version 2016
5   */
6  public class GestorPrueba {
7      /** Método principal.
8       *  @param args String[], argumentos del programa.
9       */
10     public static void main(String[] args) {
11         ItemAgenda i1 = new ItemAgenda("Ana Perez", "622115611", 46022);
12         ItemAgenda i2 = new ItemAgenda("Rosalía", "963221153", 46010);
13         ItemAgenda i3 = new ItemAgenda("Juan Duato", "913651228", 18011);
14
15         // Creación de la Agenda a1
16         Agenda a1 = new Agenda();
17         a1.insertar(i1); a1.insertar(i2); a1.insertar(i3);
18
19         // Escribir en el fichero y mostrar
20         a1.guardarAgenda("agenda1.dat");
21         System.out.println("AGENDA ALMACENADA:");
22         System.out.println(a1);
23
24         // Leer del fichero y mostrar
25         Agenda rec = Agenda.leerAgenda("agenda1.dat");
26         System.out.println("AGENDA RECUPERADA:");
27         System.out.println(rec);
28     }
29 }
```

Figura 15.22: Clase `GestorPrueba`. Creación, almacenamiento y recuperación de una agenda.

En su ejecución el programa muestra, escribiéndola en la salida estándar, la `Agenda` almacenada y recuperada, tal y como se muestra en la figura 15.23.

Si a continuación se lista el contenido del directorio en curso, se tiene:

Sistema

```
$ ls -l *.dat
-rw-r--r-- 1 profesor PRG  276 2012-04-05 18:00 agenda1.dat
```

Obsérvese que el fichero que contiene el objeto, *agenda1.dat*, ocupa 276 bytes.

Salida Estándar

```
AGENDA ALMACENADA:
Ana Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
================================
AGENDA RECUPERADA:
Ana Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
================================
```

Figura 15.23: Salida estándar tras la ejecución del programa de Agenda.

Ejemplo 15.2. Supóngase ahora que la clase `SecuenciaDeCirculos` del ejemplo 1 y las clases del paquete `figuras`, en particular, la clase `Circulo`, implementan la interfaz `Serializable`. Se quiere almacenar/recuperar en/desde un fichero binario de objetos la información de una secuencia de círculos. Para ello, se definen, en la clase `SecuenciaDeCirculos`, los métodos `guardarFormatoObjeto(ObjectOutputStream)` y `cargarFormatoObjeto(ObjectInputStream)`, que se muestran en las figuras 15.24 y 15.25, respectivamente.

El método `guardarFormatoObjeto(ObjectOutputStream)` escribe el objeto `SecuenciaDeCirculos` (el mismo objeto invocador del método) en el fichero representado por el flujo `oo`, usando el método `writeObject(Object)`. Como este método propaga la excepción `IOException`, es necesario propagarla también en `guardarFormatoObjeto(ObjectOutputStream)`.

```
/**
 * Guarda una SecuenciaDeCirculos en un fichero binario de objetos.
 * @param oo ObjectOutputStream para realizar la escritura en fichero.
 * @throws IOException si se produce un error en la escritura.
 */
public void guardarFormatoObjeto(ObjectOutputStream oo)
    throws IOException {
    oo.writeObject(this);
}
```

Figura 15.24: Método de escritura en fichero binario de objetos de tipo `SecuenciaDeCirculos`.

El método `cargarFormatoObjeto(ObjectInputStream)` lee un objeto `SecuenciaDeCirculos` almacenado en un fichero binario de objetos representado por el flujo `oi`, usando el método `readObject()`. Se recorre el array de `Circulo`

de la secuencia leída `s` y, usando el método `insertar(Circulo)`, se añaden los círculos de `s` a la secuencia `this`. La instrucción `readObject()` puede lanzar las excepciones `ClassNotFoundException` e `IOException` que se propagan para ser tratadas en la clase `GestorSecCirculos`.

```
/**
 * Carga los círculos de una SecuenciaDeCirculos desde
 * un fichero binario de objetos.
 * @param f ObjectInputStream, para realizar la lectura desde fichero.
 * @throws ClassNotFoundException si no se encuentra la clase del
 * objeto leído.
 * @throws IOException si se produce un error en la lectura.
 */
public void cargarFormatoObjeto(ObjectInputStream oi)
    throws ClassNotFoundException, IOException {
    SecuenciaDeCirculos s = (SecuenciaDeCirculos) oi.readObject();
    for (int i = 0; i < s.talla; i++) {
        this.insertar(s.elArray[i]);
    }
}
```

Figura 15.25: Método de lectura desde fichero binario de objetos de tipo `SecuenciaDeCirculos`.

Desde la clase `GestorSecCirculos`, se realizará la escritura/lectura en/desde fichero binario de objetos con los métodos `guardarSecCircBin(SecuenciaDeCirculos, String)` y `cargarSecCircBin(SecuenciaDeCirculos, String)`, respectivamente, que se muestran en las figuras 15.26 y 15.27.

El método `guardarSecCircBin(SecuenciaDeCirculos, String)`, dada una secuencia de círculos y el nombre de un fichero, después de crear un objeto `ObjectOutputStream` a partir del `FileOutputStream` del fichero, invoca al método `guardarFormatoObjeto(ObjectOutputStream)`. Se tratan, informando al usuario del error ocurrido en el proceso de escritura, las excepciones *checked*: `FileNotFoundException` (que se lanza si el objeto `FileOutputStream` no localiza el fichero que se le pasa como parámetro) e `IOException` (que puede lanzarse por el objeto `ObjectOutputStream` o ser propagada por el método `guardarFormatoObjeto(ObjectOutputStream)`). Por último, se comprueba si el `ObjectOutputStream` se ha creado y, si es así, se cierra en la cláusula `finally` del `try`. En este caso, dicho cierre requiere tratar dentro de la misma cláusula una `IOException` propagada por el método `close()`.

```
/** Guarda una SecuenciaDeCirculos en un fichero binario de objetos.
 *  @param s SecuenciaDeCirculos, de donde se guardan.
 *  @param nomF String, nombre del fichero donde escribir.
 */
private static void guardarSecCircBin(SecuenciaDeCirculos s,
    String nomF) {
    ObjectOutputStream fSal = null;
    try {
        fSal = new ObjectOutputStream(new FileOutputStream(nomF));
        s.guardarFormatoObjeto(fSal);
    } catch (FileNotFoundException e) {
        String msg = "No se puede crear el fichero " + nomF;
        System.out.println(msg);
    } catch (IOException e) {
        String msg = "No se puede escribir en el fichero " + nomF;
        System.out.println(msg);
    } finally {
        try {
            if (fSal != null) { fSal.close(); }
        } catch (IOException e) {
            String msg = "No se puede cerrar el fichero " + nomF;
            System.out.println(msg);
        }
    }
}
```

Figura 15.26: Método de escritura en fichero binario de objetos en la clase `GestorSecCirculos`.

El método `cargarSecCircBin(SecuenciaDeCirculos, String)`, dada una secuencia de círculos y el nombre de un fichero, después de crear los objetos `FileInputStream` y `ObjectInputStream`, invoca al método `cargarFormatoObjeto(ObjectInputStream)`. Se tratan, informando al usuario del error ocurrido en el proceso de lectura del fichero, las excepciones: `FileNotFoundException` (que se lanza si el objeto `FileInputStream` no localiza el fichero que se le pasa como parámetro), `IOException` (que puede lanzarse por el objeto `ObjectInputStream` o ser propagada por el método `cargarFormatoObjeto(ObjectInputStream)`) y `ClassNotFoundException` (que también puede ser propagada por el método anterior si no se puede determinar la clase del objeto que se intenta leer). Finalmente, se comprueba si el `ObjectInputStream` se ha creado y, si es así, se cierra en la cláusula `finally` del `try`. También en este caso, dicho cierre requiere tratar dentro de dicha cláusula una `IOException` propagada por el método `close()`.

```
/** Carga una SecuenciaDeCirculos desde un fichero binario de objetos.
 *  @param s SecuenciaDeCirculos, en donde se carga.
 *  @param nomF String, nombre del fichero del que leer.
 */
private static void cargarSecCircBin(SecuenciaDeCirculos s,
    String nomF) {
    ObjectInputStream fEnt = null;
    try {
        fEnt = new ObjectInputStream(new FileInputStream(nomF));
        s.cargarFormatoObjeto(fEnt);
    } catch (FileNotFoundException e) {
        String msg = "No se puede acceder al fichero " + nomF;
        System.out.println(msg);
    } catch (ClassNotFoundException e) {
        String msg = "Error de formato en la lectura de " + nomF;
        System.out.println(msg);
    } catch (IOException e) {
        String msg = "No se puede leer del fichero " + nomF;
        System.out.println(msg);
    } finally {
        try {
            if (fEnt != null) { fEnt.close(); }
        } catch (IOException e) {
            String msg = "No se puede cerrar el fichero " + nomF;
            System.out.println(msg);
        }
    }
}
```

Figura 15.27: Método de lectura desde fichero binario de objetos en la clase `GestorSecCirculos`.

Excepción `EOFException`. Determinación del final de un fichero binario

Se ha estudiado ya, en algún programa anterior, la lectura de una secuencia de datos almacenada en un fichero binario. Así, por ejemplo, en el programa visto en la sección 15.3.2, se recupera una secuencia de valores de un fichero de objetos gracias a que se conoce inicialmente el número de elementos que contiene el propio fichero.

Puede ocurrir, sin embargo, que el número de elementos del fichero no sea conocido inicialmente. Se plantea entonces la cuestión de cuándo interrumpir la lectura del fichero sin tratar de acceder más allá del final del mismo o de, al menos, recuperar adecuadamente el programa en caso de sobrepasar dicho límite.

En general, cuando en un flujo o fichero binario, bien de datos elementales (como `FileInputStream`), bien de objetos (como `ObjectInputStream`) se intente acceder más allá del final del mismo se provocará una `IOException`.

Adicionalmente, si las operaciones de lectura son de alguno de los tipos de datos elementales (tal como `readInt()`, `readDouble()`, `readBoolean()`, etc.) de las clases `FileInputStream` o `ObjectInputStream`, entonces, en el caso de intentar acceder más allá del final del fichero se producirá una excepción `EOFException`, que es subclase de `IOException`.

Gestionando adecuadamente la excepción correspondiente, es posible determinar si se ha llegado o no al final del fichero y, con ello, acabar el tratamiento. En la figura 15.28 se muestra, a título de ejemplo, el método `escribir(String)` que permite escribir un número aleatorio de valores aleatorios en un fichero, mediante un `ObjectOutputStream`, y el metodo `leer(String)` que permite leer un fichero con el formato comentado, mediante un `ObjectInputStream`, mostrando los valores almacenados en la salida estándar.

Como puede verse (líneas 51 a 61) la lectura se ha organizado mediante un bucle cuya terminación vendrá dada por el acceso al final del fichero, que provoca la `EOFException` que, correspondientemente, es tratada.

Nótese, líneas 50 a 64, que se ha anidado un bloque `try-catch` dentro de otro. Mediante el más interno (líneas 53 a 60) se gestiona el posible fin del fichero, cuya lectura se hace mediante un bucle finalizado exclusivamente por la aparición de la `EOFException`. Con el bloque más externo es posible controlar condiciones problemáticas de entrada, tales como la inexistencia del fichero, etc.

Si se ejecuta el código de la clase, se provocará una salida similar a la siguiente:

Salida Estándar

```
4  7  8  6  7  3  7  0  2  5  8  7
Final de escritura
4  7  8  6  7  3  7  0  2  5  8  7
Final del fichero
```

```
24      /** Escribe en un fichero binario de nombre fich
25       *  un número aleatorio de valores enteros aleatorios.
26       *  @param fich String, el nombre del fichero.
27       */
28      public static void escribir(String fich) {
29          try {
30              ObjectOutputStream oos =
31                  new ObjectOutputStream(new FileOutputStream(fich));
32              int alea = (int) (Math.random() * 10 + 5);
33              for (int i = 1; i <= alea; i++) {
34                  int val = (int) (Math.random() * 10);
35                  oos.writeInt(val);
36                  System.out.print(val + "  ");
37              }
38              System.out.println("\nFinal de escritura");
39              oos.close();
40          } catch (IOException fex) {
41              System.err.println("Error al guardar: " + fex.getMessage());
42          }
43      }
44
45      /** Lee de un fichero binario de nombre fich
46       *  un número aleatorio de valores enteros aleatorios.
47       *  @param fich String, el nombre del fichero.
48       */
49      public static void leer(String fich) {
50          try {
51              ObjectInputStream ois =
52                  new ObjectInputStream(new FileInputStream(fich));
53              try {
54                  while (true) {
55                      int val = ois.readInt();
56                      System.out.print(val + "  ");
57                  }
58              } catch (EOFException ef) {
59                  System.out.println("\nFinal del fichero");
60              }
61              ois.close();
62          } catch (IOException fex) {
63              System.err.println("Error al guardar: " + fex.getMessage());
64          }
65      }
```

Figura 15.28: Ejemplo de tratamiento de fin de fichero (`EOFException`).

15.6 Problemas propuestos

1. Construir un programa Java que reciba como argumento de línea de comandos la ruta a un fichero y que muestre por pantalla información básica sobre el mismo (como mínimo el nombre del fichero, directorio donde se encuentra y su tamaño expresado en kbytes).

2. Escribir un método estático que escriba en un fichero binario los números del 1 al 999.

3. Escribir un método estático que lea el fichero generado por el programa del ejercicio 2 y sume dichos números. Comprobar que el resultado es correcto implementando un bucle adicional que realice dicha suma.

4. Construir un programa que escriba en un fichero de texto los números del 1 al 999 y posteriormente los vuelva a leer de ese fichero para realizar la suma de los mismos. Verificar que el resultado es correcto. Comprobar la diferencia de tamaños entre el fichero generado en el ejercicio 2 y el generado por este ejercicio.

5. Construir un programa que permita buscar palabras en un fichero de texto. Se debe mostrar el número de línea y su contenido, para cada línea que contenga la palabra buscada.

6. Desarrollar un programa que permita eliminar todas las ocurrencias de una palabra dada en un fichero de texto. El programa recibirá como argumentos de línea de comandos la ruta al fichero así como la palabra en cuestión. Este código producirá automáticamente un nuevo fichero con la siguiente nomenclatura: Si el fichero de entrada se llama *fichero.txt*, el fichero generado se llamará *fichero_ 2.txt*.

7. Escribir un método estático que reciba como entrada el nombre de un fichero de texto y devuelva estadísticas básicas sobre el mismo (como mínimo se debe incluir el número de palabras, el número de caracteres totales del texto y la longitud media de una palabra medida en nº de caracteres).

8. Modificar el programa ejemplo de la agenda telefónica (figuras 15.20, 15.21 y 15.22) de forma que:
   - un elemento individual de la agenda (un `ItemAgenda`) mantenga además del nombre, teléfono y código postal de un contacto su dirección postal. Tras hacerlo, ¿es necesario modificar algo en las operaciones de lectura y escritura en fichero de la clase `Agenda`?
   - Escribir operaciones en la clase `Agenda` para efectuar una búsqueda de un contacto por nombre o teléfono. Ambas operaciones devolverán el primer `ItemAgenda` que cumpla la condición en caso de que exista o `null` en el caso de que no sea así.

- Crear un nuevo programa principal que, mediante el uso de un menú, permita almacenar la agenda en curso en un fichero, añadir un nuevo contacto cuyos datos se pedirán al usuario, eliminar un contacto dado su número de teléfono y, finalmente, recuperar una agenda desde un fichero dado.

9. En la clase `Agenda` (figura 15.21), se ha definido el método de clase `leerAgenda(String)` que, a partir del nombre de un fichero, devuelve el objeto `Agenda` que se encuentra almacenado en el mismo.

   Definir, utilizando el método anterior, un constructor de la clase `Agenda` que recibiendo como argumento el nombre del fichero en el que se ha almacenado el objeto `Agenda`, construya un objeto de dicho tipo. Modificar la clase `GestorPrueba` para utilizar el nuevo constructor.

10. Una estación meteorológica necesita gestionar las medidas diarias de la pluviosidad en una determinada zona a lo largo de un año con las siguientes características:

    - Se ha decidido construir una clase, denominada `Pluviometro` que tenga como atributos dos arrays, uno para almacenar el número de días de cada mes y otro para guardar las medidas de pluviosidad de dichos días. Por comodidad para el programador se ha decidido prescindir de usar la posición 0 de los arrays, para que el índice coincida con el número de mes o el número de día, de manera que las posiciones `[0]` de los arrays no se usarán.
    - `diasM` es un array de 13 `int` tal que `diasM[i]` es el numero total de días del mes `i` siendo `1 <= i <= 12`, de tal manera que como se ha comentado, `dia[0]` no se usará.
    - `lluvia` es un array bidimensional con 13 filas. `lluvia[i]` es un array de `diasM[i] + 1` valores de tipo `double` (dicho de otra manera su longitud es `lluvia[i].length == diasM[i] + 1`) tal que `lluvia[i][j]` representa la medida del día `j` del mes `i`, siendo `1 <= i <= 12` y `1 <= j <= diasM[i]`. Las posiciones `lluvia[0][j]` y `lluvia[i][0]` no se usarán.

    Se pide escribir un programa con la siguiente funcionalidad:

    a) leer los datos de pluviosidad desde un fichero *pluvio.dat* en el que cada línea tiene el siguiente formato:

    ```
    dia mes medida
    ...
    ```

    y almacenarlos en la matriz `lluvia`, validando los valores de día y mes leídos. Las medidas no tienen por qué estar ordenadas cronológicamente.

Un ejemplo de algunas líneas del fichero es el siguiente:

```
...
24 11 312.12
15 3 6.756
14 8 12.5
15 1 31.3
16 3 212.0
17 3 87.9
18 3 3.56
23 6 11.11
...
```

b) dada la matriz `lluvia` y cierto mes `m`, determinar la cantidad máxima llovida en un solo día a lo largo de dicho mes así como el día en que esta se produjo.

c) dada la matriz `lluvia`, cierto mes `m` y una cantidad `lt` de litros, determinar un día de dicho mes en que la pluviosidad haya superado dicha cantidad. Si no existe, indicarlo con un mensaje.

d) dada la matriz `lluvia` y cierto mes `m`, determinar si hubo al menos tres días consecutivos en dicho mes con una pluviosidad mayor a `100` litros cada uno de ellos.

e) mostrar por pantalla las medidas del fichero de entrada *pluvio.dat* pero ordenadas cronológicamente.

11. Se desea modificar la solución al problema anterior, de forma que la lectura de los datos se produzca de un fichero binario (mediante un `ObjectInputStream`) en lugar de un fichero de texto, tal y como se planteó antes.

    La lectura de los datos se efectuará en tríadas de valores, enteros los dos primeros, que representarán, respectivamente, el día y mes de la medida; siendo el tercer valor uno en coma flotante (un `double`) que contendrá la cantidad llovida.

    Para determinar el momento en el que se produzca el final del fichero, se **deberá utilizar** la excepción `EOFException` tal y como se menciona en el capítulo (ejemplo de la figura 15.28).

12. Se tienen los siguientes datos referentes a la última vuelta ciclista local:

    - `ciclistas`: array con los nombres de cada ciclista.
    - `tiempos`: matriz en la que en cada fila `i` se tienen los tiempos de `ciclistas[i]` en cada una de las cinco etapas, el tiempo máximo empleado en una etapa es 180 minutos (se consideran valores enteros).

    Se pide escribir un programa con la siguiente funcionalidad:

    a) diseñar la clase `VueltaCiclista` que tenga como atributos los arrays anteriormente mencionados.

b) leer los datos de un fichero de texto con el formato:

```
nº de participantes
nombre t1 t2 t3 t4 t5
otronombre t1 t2 t3 t4 t5
...
```

c) dado el nombre de un ciclista, mostrar por pantalla los tiempos empleados por este en cada una de las etapas si ha participado o el mensaje *"No ha participado en esta vuelta"* en caso contrario.

d) mostrar por pantalla el nombre del ciclista ganador de la vuelta y el tiempo que este empleó. Gana la vuelta el ciclista cuya suma de tiempos de las cinco etapas es menor.

e) mostrar por pantalla los ciclistas y sus tiempos ordenados según el tiempo empleado.

En todos los casos, los tiempos se mostrarán en horas y minutos.

13. Para resolver el problema anterior desde la *perspectiva de la programación orientada a objetos*, se plantea la siguiente organización de la información, que deberá ser implementada adecuadamente:

   - Una clase `Ciclista`, mediante la que se mantendrá información relativa a cada uno de los mismos, en particular su `nombre` así como sus `tiempos`, array de 5 elementos enteros en los que, en cada uno de ellos, se mantendrá el tiempo empleado por el ciclista en cubrir la etapa correspondiente (el tiempo máximo empleado en una etapa es 180 minutos).

     Se deberá diseñar esta clase definiendo, además, los métodos constructores, consultores y modificadores que se consideren pertinentes.

   - Una clase `VueltaCiclista` que contendrá un array `ciclistas`, de elementos de la clase `Ciclista`. Se considerará que el array tiene los elementos estrictamente necesarios; esto es, no existen posiciones del array no ocupadas.

     Se deberá diseñar esta clase definiendo, además de los métodos que se consideren pertinentes (tales como constructores, etc.), métodos para almacenar y recuperar los datos de una `VueltaCiclista` en y desde un fichero de objetos (esto es, usando `ObjectInputStream` y `ObjectOutputStream`), así como los métodos necesarios para, al igual que en el problema anterior:

     - dado el nombre de un ciclista, mostrar por pantalla los tiempos empleados por este en cada una de las etapas si ha participado o el mensaje *"No ha participado en esta vuelta"* en caso contrario.
     - mostrar por pantalla el nombre del ciclista ganador de la vuelta y el tiempo que este empleó. Gana la vuelta el ciclista cuya suma de tiempos de las cinco etapas es menor.

- mostrar por pantalla los ciclistas y sus tiempos ordenados según el tiempo empleado.

14. Si se han resuelto los dos problemas anteriores, se está en posición de discutir las mejoras (y tal vez inconvenientes) que haya podido introducir la *solución orientada a objetos*.

    Se pide señalar las diferencias más significativas entre las dos soluciones al problema de la vuelta ciclista, desde el punto de vista del almacenamiento y recuperación de la información en memoria externa. Tratar de responder a la cuestión planteada, determinando cómo la posible variación de elementos en las clases, altera la organización de los ficheros y/o de las operaciones encargadas de su lectura o escritura.

    ¿Cuál de las organizaciones de los datos parece más cómoda para trabajar si tiene que sufrir modificaciones posteriores?

15. Se desea gestionar la información sobre los visitantes a cierto parque de atracciones *Gran Aventura*:

    - de cada `visitante` se conoce sus apellidos, nombre, edad y un código como, por ejemplo, "`GA325`".
    - Existen `atracciones` que tienen cierto nombre, tales como: *Furius*, *DragonKhan*, *TutukiSplash*, *Stampida*.
    - Además, se conocen los visitantes que han participado en cada una de las atracciones ya que se mantienen los códigos de aquellos visitantes que hayan accedido a cada una de las mismas.

    Se pide escribir un programa para la gestión básica del parque de atracciones para lo que se deberá:

    a) diseñar una clase `Visitante`, para mantener los datos individuales de cada uno de ellos. La clase deberá incluir los métodos de gestión (constructores, consultores y modificadores que se consideren pertinentes).

    b) diseñar una clase `Atraccion` mediante la que se mantenga sus elementos propios, tales como su `nombre` y una lista de los visitantes que han accedido a los largo de un día a la misma (dicha lista se puede implementar mediante un array parcialmente completo de valores de tipo `Visitante`).

    Esta clase contendrá, por lo menos, dos métodos, uno para añadir un nuevo visitante a los que han accedido a la atracción y otro para determinar si dado un código de visitante, ha accedido o no a la misma,

    c) diseñar una clase `ParqueAtracciones`, mediante la que se mantenga una lista, implementada una vez más mediante un array, de las atracciones que están en funcionamiento a lo largo de un día. Esta clase deberá contener, por lo menos, los siguientes métodos:

- Métodos para almacenar y recuperar en un fichero de objetos los datos correspondientes a todo el parque de atracciones en un momento dado. Estos métodos recibirán como parámetro el nombre del fichero que contendrá los datos.
- Un constructor de la clase `ParqueAtracciones` que construirá uno de tales objetos a partir de los datos incluidos en un fichero de objetos (cuyo nombre recibirá el constructor como parámetro). Este constructor deberá utilizar el método diseñado en el punto anterior para recuperar la información de un parque de atracciones.
- Y, además, métodos para:
  - mostrar el número de visitantes que han accedido a cada una de las atracciones.
  - comprobar si un determinado visitante del que se conocen sus apellidos ha accedido a una determinada atracción.
  - mostrar los datos del `visitante` más joven de una atracción determinada.
  - mostrar por pantalla la lista de atracciones a las que ha accedido un determinado `visitante` del que se conocen sus apellidos.

Más información

[Eck15] D.J. Eck. *Introduction to Programming Using Java, Seventh Edition*. 2015. URL: `http://math.hws.edu/javanotes/`. Capítulo 11 (11.1, 11.2 y 11.3).

[Ora15] Oracle. *The Java^{TM} Tutorials*, 2015. URL: `http://download.oracle.com/javase/tutorial/`. Trail: Essential Java Classes. Lesson: Basic I/O.

[SM16] W.J. Savitch. *Absolute Java, Sixth Edition*. Pearson Education, 2016. Capítulo 10.

[Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007. Capítulo 10.