

UD07.1. Ficheros de Texto.

Apuntes

DAM1-Programación 2024-25 (Parainfo Capítulo 10)

Introducción	2
1. Excepciones	3
2. Flujos de entrada de texto	12
3. Scanner y flujos de entrada	15
4. Flujos de salida de texto	18
Apertura con Recursos (try-catch-resources)	19
5. Ficheros XML y Java. API JAXB	20

Otras fuentes:

- [Programación Java: Teoría](#)
 - [La Clase File](#)
 - [Ficheros de Texto](#)
 - [Ficheros Binarios](#)
 - [Ficheros de Acceso Aleatorio](#)
 - [Persistencia y Serialización de objetos](#)
 - [Seleccionar archivos de forma gráfica con JFileChooser](#)
- [Píldoras Informáticas: Curso de Java desde 0](#)
 - (Vídeos 152 a 160)
- [w3schools Java Tutorial](#)
 - [Java Files](#)

Introducción

En la mayoría de los programas, en un momento u otro hay que interaccionar con alguna fuente o soporte de datos, como un archivo en un dispositivo de almacenamiento de datos o un dispositivo de red, ya sea para guardar información o para recuperarla. Java implementa una serie de clases llamadas *flujos*, encargadas de comunicarse con estos dispositivos. El funcionamiento de estos flujos, desde el punto de vista del programador, no depende del tipo del dispositivo hardware con el que está asociado, lo que nos liberará del trabajo que supone tener en cuenta las características físicas de cada uno de ellos.

Los flujos pueden ser de entrada o de salida, según sean para recuperar o guardar información. Por otra parte, atendiendo a la clase de datos que se transmiten, los flujos son de dos tipos:

- **Carácter:** si se asocian a archivos u otras fuentes de texto.
- **Binarios:** si transmiten bytes, cuyos valores están comprendidos entre 0 y 255. Estos, en realidad, permiten manipular cualquier tipo de datos.

Vamos a empezar por los flujos de tipo texto, mientras que en la siguiente unidad nos dedicaremos a los archivos —flujos— de tipo binario. Pero, dado que en los flujos de datos entre el ordenador y los dispositivos de almacenamiento se producen errores frecuentes, como intentos de acceso a ficheros inexistentes o con nombres mal escritos, tendremos que estudiar las **excepciones**, que es como se llaman los errores en Java.


Argot técnico



Un flujo (*stream*, en inglés) es una abstracción que permite que un programa se conecte con un dispositivo físico (un disco duro, un puerto de red, un DVD, etc.) para recibir o enviar información.

No debemos confundir los flujos de datos con los objetos de tipo *Stream*, que estudiaremos en una unidad posterior.

1. Excepciones

-  UD03.Excepciones en Java
- [Programación Java: Excepciones. Bloques try, catch, finally.](#)
- [Programación Java: Excepciones en Java con Ejemplos](#)

Los errores en los programas son prácticamente inevitables, tanto si están originados por códigos deficientes, entrada de datos, parámetros incorrectos o archivos inexistentes como si lo están por discos defectuosos, entre otros.

En la mayoría de los lenguajes de programación, la manipulación de los errores suele ser complicada y confusa. Su código se mezcla con el del resto del programa, haciéndolo poco claro, y suele estar destinado solo a evitar el error y no a controlar la situación una vez que dicho error se ha producido.

Java es un lenguaje que se adapta a las condiciones de internet, y sus programas se ejecutarán en máquinas remotas, casi siempre manipuladas por personal no cualificado.

Esto obliga a adoptar un enfoque nuevo, en el que se trata de evitar, en lo posible, que el programa se interrumpa a causa del error. Para ello no basta con evitar que se produzcan los errores, sino que hay que implementar los medios para que un programa se recupere de las condiciones generadas por un error que no se ha podido evitar. Eso depende del tipo de error, y no siempre es posible.

Cuando, en la ejecución de un programa, se produce una situación anormal —un error— que interrumpe el flujo normal de ejecución, el método que se esté ejecutando genera un objeto de la clase `Throwable` («arrojable»), que contiene información del error, de su causa y del contexto del programa en el momento en que se produce, y lo entrega (lo «arroja») al sistema de tiempo de ejecución —la máquina virtual—. Este objeto es susceptible de ser capturado —ya veremos cómo— por el programa y analizado para dar una respuesta, si procede. En caso contrario, el programa se interrumpe y el sistema de tiempo de ejecución muestra una serie de mensajes que describen el error, como ocurre en otros lenguajes de programación.

Hay errores lo bastante graves para que sea preferible que el programa se interrumpa. Por ejemplo, errores derivados de problemas de hardware. Si intentamos leer de un disco defectuoso, se producirán errores de los que es difícil, si no imposible, recuperarse. Estos y otros errores relacionados directamente con la máquina virtual, y no con el código de nuestro programa, arrojan objetos de la clase `Error`, una subclase de `Throwable`. De ellos no nos vamos a ocupar, ya que poco se puede hacer con estos errores a la hora de programar. Tampoco nos ocuparemos de las excepciones llamadas *de tiempo de ejecución* (`runtime exceptions`), que proceden de líneas de código equivocadas. La solución a estos errores es corregir el código.

Pero hay otra clase de errores más habituales y menos graves, como entradas de datos de tipo equivocado, aperturas de ficheros con ruta de acceso errónea u operaciones aritméticas no permitidas. A estos errores se les llama *excepciones*, y producen un objeto de la clase `Exception`, otra subclase de `Throwable`. A continuación explicaremos estas excepciones, ya que son manipulables a través del código. Para ello se usan los bloques `try`, `catch` y `finally`.

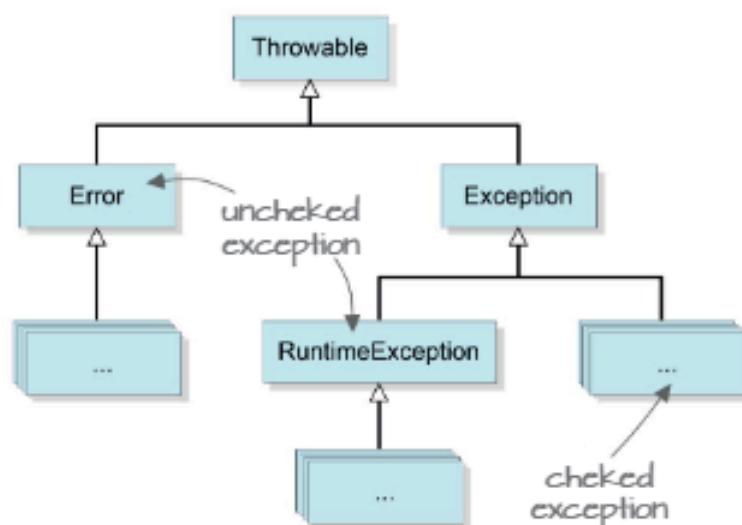


Figura 10.1. Estructura de las clases que forman el sistema de excepciones de Java.

Cuando sabemos que en un determinado fragmento de código se puede producir una excepción, lo encerramos dentro de un bloque `try`. Por ejemplo, si en una división entre las variables `a` y `b` sospechamos que el divisor `b` podría hacerse cero en algún momento, escribimos

```
try {  
    int c;  
    c = a / b;  
}
```

Con esta estructura estamos sometiendo a observación al bloque de código encerrado entre llaves. Si salta una excepción en ese bloque, deberá ser capturada por un bloque `catch` de la siguiente forma:

```
catch(ArithmeticException e) {  
    System.out.println("Error: división por cero");  
}
```

Si se produce la excepción y es capturada, se interrumpe la ejecución del código del bloque `try`, saltando a la primera línea del bloque `catch`. Cuando se termina de ejecutar dicho bloque, continúa en la línea inmediatamente posterior a la estructura `try-catch`.

La palabra reservada `catch` va seguida de unos paréntesis donde se encierra un parámetro de la clase de excepción que puede atrapar; en este caso, una excepción aritmética. El parámetro `e` se puede usar como variable local dentro del bloque. Hace referencia al objeto de la excepción y contiene toda la información sobre el error que la ha producido. Entre sus métodos está `getMessage()`, que nos muestra un mensaje descriptivo del error. Podríamos haber puesto

```
catch(ArithmeticException e) {  
    System.out.println(e.getMessage());  
}
```

o incluso

```
catch(ArithmeticException e) {  
    System.out.println(e);  
}
```

que hace una llamada a `toString()` de la clase de la variable `e`, donde también se describe el error (en inglés).

Los bloques `try` y `catch` deben ir uno a continuación del otro, sin ningún código en medio.

```
try {  
    //_bloque try  
} catch(TipoExcepción nombreParámetro) {  
    //_bloque catch  
}
```

En el bloque `catch` solo se recogerán las excepciones del tipo declarado entre paréntesis o de una subclase. En el ejemplo anterior podríamos haber escrito lo siguiente:


```

try {
    c = a / b;
} catch(Exception e) {
    System.out.println("Error: división por cero");
}

```

ya que `ArithmeticException` es una subclase de `Exception`.

Esto nos permitiría recoger otros tipos de excepción en el mismo `catch`, pero, para ese caso, es mejor escribir más de un bloque `catch`. Con un mismo bloque `try` se pueden poner tantos bloques `catch` como deseemos, siempre que vayan seguidos,

```

try {
    //...bloque try
} catch(tipoExcepción1 nombreParámetro1) {
    //...bloque catch
} catch(tipoExcepción2 nombreParámetro2) {
    //...bloque catch
} ...

```

Cuando se produce una excepción en el bloque `try`, se compara con el tipo del primer bloque `catch`. Si coincide con él o con una subclase, se ejecuta dicho bloque y continúa el programa después del último bloque `catch`. Si no coincide, se compara con el tipo del segundo bloque, y así sucesivamente hasta que se encuentra un bloque cuyo parámetro coincida o sea una superclase de la excepción producida, de forma que solo se ejecuta un bloque `catch`, el primero cuyo tipo sea compatible. Si la excepción no coincide con ninguno de los parámetros de los bloques, no es capturada y se interrumpe la ejecución del programa. Aquí hay que tener cuidado de no poner un bloque `catch` con una excepción que sea superclase de otra que vaya más abajo, pues el bloque de esta última nunca se ejecutará.

Por ejemplo:

```

try {
    c = a / b;
} catch(Exception e) {
    System.out.println("Estoy en el primer catch");
} catch(ArithmeticException e) {
    System.out.println("Estoy en el segundo catch");
}

```

Si `b` vale cero, se producirá una excepción de tipo `ArithmeticException`, pero, al ser un subtipo de `Exception`, será capturada en el primer `catch`, cuyo bloque será el que se ejecute, apareciendo el mensaje: «Estoy en el primer `catch`». La ejecución seguirá después del último bloque, de modo que el segundo bloque `catch` es inútil, ya que jamás se va a ejecutar. De hecho, en nuestro ejemplo —como todo tipo de excepción es subclase de `Exception`— cualquier excepción que se produzca en el bloque `try` será capturada en el primer bloque `catch` y ningún bloque que pongamos después se va a ejecutar nunca.

Por otra parte, existe la posibilidad de capturar más de un tipo de excepción con un único bloque `catch`.

```

catch(tipoExcepción1 | tipoExcepción2 | ... e) {
    ...
}

```

Aquí, la barra vertical equivale a una disyunción lógica **O**. Se pueden añadir tantos tipos de excepción como se quiera, separados por barras verticales. El significado es: «si se arroja una excepción del tipo `tipoExcepción1` o del tipo `tipoExcepción2` o `...`, ejecutar este bloque `catch`, donde la excepción será referenciada con la variable `e`».

Todo código en Java forma parte de algún método que, en última instancia, puede ser el método `main()`. Si desde un método `metodo2()` se invoca otro método `metodo1()` y salta una excepción durante la ejecución de este último, podemos capturarla y manipularla, como hemos hecho hasta ahora, con una estructura `try-catch` en el propio código de `metodo1()`. Pero hay un enfoque alternativo. Podemos declarar, en la definición de `metodo1()`, que en su ejecución puede producirse dicha excepción. Para ello usaremos la palabra clave `throws`.

```

tipo metodo1(tipo1 parametrol,...) throws tipoExcepción {
    ...
}

```

En el cuerpo de `metodo1()` ya no tenemos que insertar los bloques `try-catch` para ese tipo de excepción. En cambio, `metodo2()` será el encargado de gestionarla cuando invoque a `metodo1()`.

```

tipo metodo2(tipo1 parametrol, tipo2 parametro2...) {
    ...
    try {
        metodo1(); //llamada al metodo1()
    } catch (tipoExcepción e) {
        //...tratamiento de la excepción
    }
}

```

Veámoslo con un ejemplo. Supongamos que en método `metodo1()` se puede dar una división por cero. Por otra parte, un segundo método `metodo2()` llama a `metodo1()`. Lo que hemos hecho hasta ahora es:

```

void metodo1(int a, int b) {
    int c;
    try {
        c = a / b;
    } catch(ArithmeticException e) {
        System.out.println("División por cero");
    }
    System.out.println("a/b = " + c);
}

```

`metodo2()`, que llama a `metodo1()`, podría ser:

```

void metodo2() {
    int x, y;
    ...
}

```

```

        metodol(x, y);
        ...
    }

```

Si la variable `y` es cero, la excepción producida es capturada y manipulada en el lugar donde se ha intentado la división —en `metodol()`— antes de que la ejecución vuelva al método que lo llama —`metodo2()`—.

Pero podríamos hacerlo de otra forma. Primero, eliminamos el bloque `try-catch` de `metodol()` y declaramos a este último como susceptible de producir una `ArithmeticException`. Esto se implementa por medio de la palabra clave `throws` en su encabezamiento,

```

void metodol(int a, int b) throws ArithmeticException {
    int c;
    c = a / b;
    System.out.println("a/b = " + c);
}

```

Con esto estamos declarando que, dentro del método, puede producirse una excepción aritmética y que deberá ser manipulada por código externo, en el método que llame a `metodol()`, en nuestro caso `metodo2()`. Además, esta particularidad, que forma parte de la definición de `metodol()`, deberá constar en la documentación que la acompaña para que los usuarios del método sepan a qué atenerse. La implementación de `metodo2()`, por tanto, deberá hacerse cargo de la excepción,

```

void metodo2() {
    int x, y;
    ...
    try {
        metodol(x, y);
    } catch(ArithmeticException e) {
        System.out.println("División por cero");
    }
    ...
}

```

Por supuesto, `metodo2()` podría «pasar» la excepción a un tercer método que lo invoque, y así sucesivamente.

Una estructura `try-catch` supone una bifurcación en el programa. Pero a menudo estamos interesados en que una serie de líneas de código se ejecuten, tanto si se produce una excepción como si no; por ejemplo, para cerrar un archivo en el que estábamos escribiendo.

Para esto se define el bloque opcional `finally` que, cuando está presente, se coloca al final de una estructura `try-catch` —es posible una estructura `try-finally`, sin bloque `catch`— y se ejecuta independientemente de si en el bloque `try` se ha lanzado una excepción o no, y de si la excepción ha sido capturada o no. Se ejecutará antes incluso que cualquier sentencia `return` que aparezca en los bloques `try` o `catch`. Esto nos asegura que, en circunstancias anómalas, se van a ejecutar determinadas tareas, como cerrar ficheros abiertos, antes de que termine la ejecución del programa. En el siguiente trozo de código:


```

try {
    //...bloque que trabaja con archivos
    return
} catch(IOException e) {
    //...bloque si salta excepción
} finally {
    //...código para cerrar archivos
}
...
return;

```

el bloque `finally` se ejecuta incluso antes de ejecutarse el `return` del bloque `try`, a pesar de que figura después de dicha sentencia, tanto si se produce una excepción como si no.

10.1.1. Requisito de captura o especificación

Al principio distinguimos entre las excepciones —clase `Exception`— y los errores propiamente dichos —clase `Error`—. Pero, entre las excepciones, hay un grupo especialmente importante, ya que son predecibles a partir del código y es fácil recuperarse de ellas. Tanto es así que el propio compilador sabe dónde se pueden producir y nos obliga a manipularlas, ya sea por medio de estructuras `try-catch` o declarándolas en el encabezamiento de los métodos (mediante `throws`).

Este grupo de excepciones, llamadas **excepciones comprobadas** (*checked exceptions*), entre las que se encuentran las más comunes, generalmente con un origen fuera del programa (entradas de datos, nombres de ficheros incorrectos, etc.), se dice que están sometidas al requisito de «capturar o especificar» (*catch or specify*); es decir, o implementamos el bloque `try-catch`, o especificamos la excepción en la declaración del método por medio de `throws`. Como el compilador nos exige su tratamiento (si no lo hacemos, genera un error de compilación), no tenemos que preocuparnos por saber cuáles son exactamente, aunque con la práctica acabamos familiarizándonos con las más importantes: `IOException`, `FileNotFoundException`, `NumberFormatException`, `ClassCastException`, etcétera.

Junto a ellas se encuentran las excepciones **no comprobadas** (*unchecked exceptions*), como `ArithmeticException`, producidas por errores aritméticos, o `ArrayIndexOutOfBoundsException`, que se produce cuando intentamos salirnos de los límites de una tabla. Dichas excepciones suelen estar asociadas a malas prácticas de programación. Por tanto, más que tratarlas con una estructura `try-catch`, hay que corregir el código para evitar que se produzcan.

Argot técnico



Se llaman *excepciones comprobadas* o *checked exception* a aquellas cuya posible ocurrencia es predecible y, por tanto, anticipable por el compilador, que nos exige su tratamiento por medio de una estructura `try-catch` adecuada o por la cláusula `throws`.

Ejemplo de excepciones no comprobadas.

```
public static void main(String[] args) {
    try {
        // Bloque de código susceptible de provocar una excepción
        System.out.println("EJEMPLOS DE EXCEPCIONES NO COMPROBADAS:");
        Scanner sc = new Scanner(System.in);
        System.out.println("(1) - Provoca ArithmeticException");
        System.out.println("(2) - Provoca ArrayIndexOutOfBoundsException");
        System.out.println("(3) - Provoca NullPointerException");
        System.out.println("(no entero) - Provoca InputMismatchException");
        System.out.print("Elige una opción: ");

        int opcion = sc.nextInt();
        switch (opcion) {
            case 1: int a = 5/0; // División por cero
            case 2: int[] arr = new int[0]; arr[1] = 0; // Índice fuera del límite
            case 3: Integer i = null; i.compareTo(5); // Acceso a un miembro de null
        }
    } catch (Exception e){
        // Se puede capturar cada excepción con su propio bloque catch() o
        // capturarlas todas indicando la clase "madre" Exception.
        System.out.println("\nEXCEPCIÓN!!");
        System.out.println("toString(): " + e);
        System.out.println("Clase: " + e.getClass().getSimpleName());
        System.out.println("Mensaje: " + e.getMessage());
    } finally {
        // Este bloque se ejecutará siempre
        System.out.println("FIN DEL EJEMPLO");
    }
}
```

E1001. Escribir el método

```
Integer leerEntero()
```

que pide un entero por consola, lo lee del teclado y lo devuelve. Si la cadena introducida por consola no tiene el formato correcto, muestra un mensaje de error y vuelve a pedirlo.

10.1.2. Excepciones de usuario

Hasta ahora, todas las excepciones que hemos visto vienen predefinidas en Java. Pero podemos implementar las nuestras propias con tan solo heredar de alguna que ya exista. Además, es posible lanzarlas cuando nos interese por medio de la palabra reservada `throw`.

Por ejemplo, si estamos introduciendo por teclado un valor entero que representa una edad, no tiene sentido un número negativo. Normalmente, en estos casos nos conformamos con un condicional y un simple mensaje de error. Pero también podemos crear con `new` y lanzar con `throw` una excepción definida por nosotros.

En la definición de una excepción de usuario no necesitamos implementar ningún método; basta con heredar de `Exception`. En todo caso, podemos sustituir el método `toString()` por uno que represente mejor nuestro caso.

```
public class ExcepcionEdadNegativa extends Exception {
    public String toString(){
        return "Edad negativa";
    }
}
```

Veamos un ejemplo donde se usa esta excepción:

```
try {
    System.out.print("Introducir edad: ");
    int edad = new Scanner(System.in).nextInt();
    if (edad < 0) {
        throw new ExcepcionEdadNegativa();
    } else {
        //cualquier código donde se use edad, por ejemplo:
        System.out.println("edad correcta: " + edad);
    }
} catch (ExcepcionEdadNegativa ex) {
    System.out.println(ex);
}
```

Clases del paquete [java.io](#)

- [File](#)
- [InputStreamReader](#)
 - [FileReader](#): `read()`, `close()`
 - [BufferedReader](#): `readLine()`
- [OutputStreamWriter](#)
 - [FileWriter](#)
 - [BufferedWriter](#): `write(int c)`, `write(String)`, `newLine()`, `flush()`, `close()`
- [FileInputStream](#): puede abrir un fichero como un flujo de datos para proporcionar a una clase `Scanner` para su tratamiento.

Clases del paquete [java.util](#)

- [Scanner](#): `next()`, `nextInt()`, `nextLine()`, etc.

2. Flujos de entrada de texto

Los flujos de entrada de tipo texto heredan de la clase `InputStreamReader`. Todas las clases que vamos a usar para trabajar con ficheros se encuentran en el paquete `java.io`. Podemos importarlas todas con la sentencia,

```
import java.io.*;
```

Las clases de entrada de texto tienen siempre un nombre que termina en `Reader`. Nosotros usaremos flujos del tipo `FileReader`. El constructor es:

```
FileReader(String nombreArchivo)
```

al que se le pasa, como parámetro, el nombre del archivo al que se quiere asociar un flujo para su lectura. Este nombre puede llevar incluida la ruta de acceso si el archivo no está en la carpeta de trabajo. Por ejemplo,

```
FileReader in = new FileReader("C:\\programas\\prueba.txt");
```

crea un flujo de texto asociado al archivo *prueba.txt*, que se halla en la carpeta *programas* de la unidad C. No hay que olvidar que, para imprimir la barra invertida, hay que escribirla doble (`\\`), ya que escrita de forma simple se emplea para las secuencias de escape (por ejemplo, `«\n»` significa un cambio de línea). Cuando estamos trabajando en una plataforma Linux, las rutas de acceso son distintas. Por ejemplo,

```
FileReader in = new FileReader("/home/pedro/programas/prueba.txt");  
//la barra hacia delante (/) se puede escribir simple
```

La apertura de un fichero puede arrojar una excepción del tipo `IOException` —Input-Output Exception, excepción de entrada y salida— o alguna subclase, cuando el archivo no se abre por alguna razón. Por ejemplo, el constructor de `FileReader` puede arrojar una excepción `FileNotFoundException`, que hereda de `IOException` si el archivo que queremos abrir para lectura no existe, o no se encuentra en la ruta de acceso especificada. Por ello, dicha operación siempre deberá ir dentro de la estructura `try-catch` correspondiente. Cuando se abre un archivo, un cursor se posiciona al principio, apuntando al primer carácter, e irá avanzando por el archivo a medida que vayamos leyendo de él.

Una vez que se ha abierto el fichero para lectura, podremos leer del flujo asociado, usando el siguiente método:

- `int read()`: lee del fichero y devuelve un carácter *Unicode* codificado como un entero. Si queremos recuperar el carácter que lleva dentro cada entero, debemos aplicarle un cast, ya que la conversión, al ser de estrechamiento, no es automática. Sabremos que hemos llegado al final del archivo cuando `read()` devuelva `-1`, que no corresponde a ningún carácter.

A medida que vamos llamando al método `read()`, el cursor va avanzando dentro del archivo, apuntando al siguiente carácter. Una vez que terminemos de leer del flujo, hay que cerrarlo con el siguiente método:

- `void close()`: cierra el flujo de entrada con objeto de completar las lecturas pendientes y liberar el archivo.

Pongamos un ejemplo: queremos leer el archivo de texto `Main.java` de uno de los proyectos que ya hemos terminado. Lo copiamos de su ubicación original y lo pegamos en la carpeta del proyecto actual que estamos implementando en NetBeans (la carpeta de trabajo, que lleva el mismo nombre que nuestro proyecto), al lado de los archivos `build.xml`, `manifest.mf`, etc. Con esta ubicación, al crear el flujo de entrada, basta con poner el nombre del archivo, sin ruta de acceso.

```
FileReader in = null;
try {
    in = new FileReader("Main.java");
    ...
} catch(IOException ex) {
    System.out.println(ex.getMessage());
}
```

E1002. Leer el archivo de texto `Main.java` de uno de los proyectos que hayamos terminado y mostrarlo por pantalla.

* Cuando usamos una ruta relativa en los ficheros se usa la carpeta del proyecto como carpeta raíz.

`FileReader` nos permite leer cualquier archivo de texto plano creado con un editor de texto como NotePad o Gedit (no con un procesador de texto, como Word).

Sin embargo, por razones de eficiencia, no se suele usar `FileReader` tal cual. Normalmente se emplean flujos de la clase `BufferedReader`, que no es más que un `FileReader` filtrado para asociarle un búfer (espacio reservado para almacenamiento temporal) en memoria. Esto permite hacer lecturas en el dispositivo físico (el disco, por ejemplo) de grupos de caracteres, en vez de caracteres individuales. Estos son colocados en cola en el búfer, a la espera de que el programa los vaya reclamando. Con ello se reduce el número de accesos al disco, que es una operación extremadamente lenta. Para crear un `BufferedReader`, basta pasarle al constructor un objeto `FileReader`.

```
BufferedReader in = new BufferedReader(new FileReader("Main.java"));
```

Ahora, el flujo `in` dispone del método `read()` para hacer las lecturas de caracteres individuales, pero, además, al tener un búfer asociado, puede hacer lecturas de líneas completas con el siguiente método:

- `String readLine()`: lee y devuelve líneas completas del archivo de texto, sin incluir el cambio de línea del final. Al llegar al final del fichero devuelve `null`.

El ejemplo anterior lo podríamos implementar ahora con un flujo de tipo `BufferedReader`.

E1003. Repetir la Actividad resuelta E1002 usando un flujo de texto con búfer.

Actividad propuesta 10.1

Crea un fichero de texto con un editor e implementa un programa que lo abra y lo lea, mostrando su contenido por pantalla.

Nota: No olvides escribir la ruta de acceso al fichero completa si este no se encuentra en la carpeta del proyecto actual.

E1004. Crear con un editor el fichero de texto `NumerosReales.txt` en la carpeta del proyecto actual y escribir en él una serie de números reales separados por espacios simples.

Implementar un programa que acceda a `NumerosReales.txt`, lea los números y calcule la Suma y la media aritmética, mostrando los resultados por pantalla.

3. Scanner y flujos de entrada

Hasta ahora hemos usado la clase `Scanner` para leer datos introducidos por el teclado, pero este, en realidad, es un flujo de entrada de texto (`System.in`). Lo que hace la clase `Scanner` es acceder al búfer del teclado en busca de secuencias de caracteres —llamadas *tokens*— que se adapten al tipo de datos requeridos por el método invocado: `next()`, `nextInt()`, etc. Esto supone que `Scanner`, además de leer secuencias de caracteres, es capaz de analizarlas y convertirlas en datos de tipos primitivos. Por defecto, para que un objeto `Scanner` identifique los distintos *tokens*, estos deben estar separados por caracteres blancos —espacios, tabuladores o cambios de línea— o secuencias de ellos. Por ejemplo, si queremos leer del teclado una serie de cinco números enteros, y guardarlos en una tabla, podemos escribir

```
int[] tabla=new int[5];
Scanner s = new Scanner(System.in);
System.out.print("Introducir serie de 5 enteros: ");
for (int i = 0; i < 5; i++) {
    int n = s.nextInt();
    tabla[i]=n;
}
System.out.println(Arrays.toString(tabla));
```

Al ejecutar el programa, es posible introducir los números de uno en uno, pulsando Intro cada vez, o podemos escribirlos todos separados por espacios en una sola línea —por ejemplo: 1 34 22 0 143— y pulsar Intro después. En ambos casos obtendremos por pantalla la serie de los números introducidos:

```
[1, 34, 22, 0, 143]
```

Si hemos introducido todos los números en una misma línea, en cada llamada a `nextInt()`, el `Scanner` localiza los espacios separadores y reconoce cada *token* (grupo de caracteres no blancos), lo analiza, identifica un entero, lo convierte en un dato de tipo `int` y lo devuelve. La sentencia de asignación correspondiente lo asigna a la variable `n`.

Argot técnico



En una secuencia de caracteres leída con un objeto `Scanner`, llamamos *token* o *componente léxico* a una subsecuencia separada del resto por secuencias separadoras que, por defecto, son caracteres o grupos de caracteres blancos. Las secuencias separadoras se pueden cambiar por otras o por algún patrón definido en una *expresión regular*. Las expresiones regulares se salen del objetivo de este libro, aunque tienen una gran importancia en programación.

Actividad propuesta 10.2

Pide por teclado el nombre, la edad (`int`) y la estatura en metros (`double`) de un deportista. Introduce los datos en una sola línea y léelos con un objeto `Scanner`. Muestra los resultados por pantalla.

Recuerda



Si queremos usar el punto decimal en vez de la coma al leer un número real con `Scanner`, debemos establecer como localización un país que la use, como Estados Unidos.

```
Scanner s = new Scanner(System.in).useLocale(Locale.US);
```

Igual que hemos usado `Scanner` para leer y analizar una secuencia de caracteres tecleados en una línea, podemos analizar una cadena de caracteres. En este caso no leeremos del flujo procedente del búfer del teclado (`System.in`), sino de un objeto `String`, que le pasaremos al constructor de `Scanner`.

```
String numeros = "1 34 22 0 143";  
Scanner s = new Scanner(numeros);
```

El resto del programa sería igual y produciría los mismos resultados.

Ahora podemos usar la clase `Scanner` para analizar el contenido de un archivo de texto. Lo abrimos creando un flujo de tipo `BufferedReader` y analizamos las cadenas devueltas por el método `readLine()`, línea a línea.

E1005. Crea con un editor un archivo de texto con un conjunto de números reales, **uno por línea**. Guarda el fichero en la misma carpeta que el código fuente.

Crea un programa que abra el fichero con un flujo de texto para lectura y que lo lea línea a línea. Utiliza la clase `Scanner` para convertir las cadenas leídas en números de tipo `double`, y mostrar al final la suma de todos ellos.

Prueba:

- Y si tenemos más de un número por línea?
- Y si en las líneas también hay textos u otros datos?
- Y si colocamos el fichero en otra ubicación?

Además de al flujo de texto introducido por teclado, la clase `Scanner` puede acceder a otros flujos de texto, como por ejemplo el asociado a un archivo de texto, con su ruta de acceso. La clase `Scanner` es útil cuando un archivo de texto contiene tokens que representan datos numéricos que hay que identificar y codificar como números para luego hacer cálculos.

E1006. Crea con un editor el fichero de texto `Enteros.txt` en la carpeta del proyecto actual y escribir en él una serie de enteros separados por secuencias de espacios y tabuladores, incluso en líneas distintas, tal como:

```
2   3   45           73  
  
123  4 21
```

Implementar un programa que acceda a Enteros.txt con un objeto Scanner a través de un flujo de entrada, lea los números y calcule su suma y su media aritmética, mostrando los resultados por pantalla.

¿Qué pasará si una línea contiene datos alfanuméricos intercalados con los números enteros?

Piensa como modificar el programa para que se salte los datos no enteros que pueda encontrar y sí procese los siguientes números.

Muestra por consola los valores que no correspondan a números enteros y contabiliza cuántos de estos “errores” tiene el fichero.

AP103_Jugadores

Crea con un editor el fichero de texto Jugadores.txt y escribe en él los nombres, edades y estaturas de los jugadores de un equipo, cada uno en una línea.

Juan 22 1.77
Luis 22 1.80
Pedro 20 1.73
...

Implementa un programa que lea del fichero los datos, y genere e imprima una lista de objetos de tipo Jugador, calcule la media de la edad y de las estaturas.

4. Flujos de salida de texto

Si en vez de leer de un archivo de texto queremos escribir en él, necesitaremos un flujo de salida de texto. Para ello, crearemos un objeto de la clase `FileWriter`, que hereda de `OutputStreamWriter` —las clases de salida de texto tienen un nombre que acaba en `Writer`—.

Los constructores de `FileWriter` son:

```
FileWriter(String nombreArchivo )
FileWriter(String nombreArchivo , boolean append)
```

donde `nombreArchivo`, como ya ocurría en `FileReader`, puede contener la ruta de acceso. El primer constructor destruye la versión anterior del archivo y escribe en él desde el principio.

Sin embargo, el booleano `append`, cuando vale `true`, nos permite añadir texto al final del archivo, respetando el contenido anterior. La apertura de un `FileWriter` puede generar una excepción del tipo `IOException`, que habrá que tratar con el `try-catch` correspondiente.

Como hicimos con `FileReader`, para mejorar el rendimiento usaremos una versión con búfer, `BufferedWriter`, que guarda los caracteres en un búfer. Cuando este está lleno, se graban en la unidad de almacenamiento correspondiente. Al constructor de `BufferedWriter` se le pasa como parámetro un flujo de salida de tipo `FileWriter`. Por ejemplo,

```
BufferedWriter out;
out = new BufferedWriter(new FileWriter("salida.txt"));
```

Los métodos de que disponemos con `BufferedWriter` son:

- `void write(int caracter)`: escribe un carácter en el archivo.
- `void write(String cadena)`: escribe una cadena en el archivo.
- `void newLine()`: escribe un salto de línea en el fichero. Se debe evitar el uso explícito del carácter «`\n`» para insertar saltos de líneas, ya que su codificación es distinta según la plataforma utilizada.
- `void flush()`: vacía el búfer de salida, escribiendo en el fichero los caracteres pendientes.
- `void close()`: cierra el flujo de salida, vaciando el búfer y liberando el recurso correspondiente.

E1007. Crea un programa que guarde en un fichero el texto “Quijote.txt”,

"En un lugar de La Mancha,
de cuyo nombre no quiero acordarme"

Escribe la primera línea en el fichero carácter a carácter, y la segunda, en una sola sentencia.

Actividad propuesta 10.4

Escribe un texto en un archivo de texto, línea a línea leídas del teclado, hasta que introduzca la cadena “fin”, excluyendo esta palabra.

Apertura con Recursos (*try-catch-resources*)

Es un error común olvidarse de cerrar el flujo, con lo que podemos encontrarnos un archivo vacío porque no todos los caracteres del búfer se han llegado a escribir en el fichero. Por eso es importante utilizar el método `close()` en el bloque `finally` o en algún otro punto donde se ejecute con seguridad.

Sin embargo, a partir de Java 7 disponemos de la estructura *try-catch-resources* o **apertura con recursos** que facilita cerrar archivos y liberar recursos sin necesidad de usar el método `close()` ni el bloque `finally`. La sintaxis consiste en declarar e iniciar el flujo a continuación del `try` entre paréntesis. Por ejemplo:

```
try (BufferedWriter out = new BufferedWriter(new FileWriter("Quijote.txt"))) {  
    String linea = "En un lugar de La Mancha,";  
  
    // Escribe la línea caracter a caracter;  
    for (int i = 0; i < linea.length(); i++)  
        out.append(linea.charAt(i));  
  
    // Escribe el salto de línea  
    out.newLine();  
  
    linea = "de cuyo nombre no quiero acordarme";  
    // Escribe una línea completa  
    out.append(linea);  
}  
catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

Tanto si se produce la excepción como si no, el flujo `out` se cierra automáticamente al terminar de ejecutarse la estructura `try-catch`. En el paréntesis que sigue a `try` se pueden abrir varios flujos separando las sentencias de apertura con un punto y coma “;”.

E1008. Escribir un programa que duplique el contenido de un fichero cuyo nombre se pide al usuario. El fichero copia tendrá el mismo nombre con el prefijo “copia_de_”.

Utiliza `FileChooser` de `JavaFX` para que el usuario seleccione el fichero.

5. Ficheros XML y Java. API JAXB

- [XML data binding - Wikipedia](#)
- [JAXB - Wikipedia](#)
- [JAXB](#)
 - [Download standalone distribution](#)
 - `\jaxb-ri\mod\jaxb-api.jar`

Una forma cada vez más común de guardar y transmitir información es por medio de los ficheros de texto de tipo XML. Como consecuencia, se presenta el problema de procesar dicha información con programas escritos en lenguajes de programación, como Java. Esto obliga a traducir la información contenida en dichos ficheros a valores de tipos definidos en los distintos lenguajes para su procesamiento y viceversa. Para dar respuesta a todo esto, Java ha implementado la API JAXB (Java Architecture for XML Binding), que permite enlazar los elementos de un archivo XML con un conjunto de clases y objetos de Java, creados con ese fin, haciendo las conversiones en los dos sentidos. Nosotros vamos a hacer una pequeña introducción con un ejemplo sencillo —un estudio completo se saldría de los objetivos de este libro; para el lector que esté interesado, se recomienda la documentación original de Java con sus tutoriales—.

Supongamos que tenemos guardados los datos de los socios de un club en archivos XML separados, uno por cada socio. Por ejemplo, los datos de Martin Fisher, en el fichero *socio.xml*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<socio id="23">
  <nombre>Martin Fisher</nombre>
  <direccion>43 Bass St</direccion>
```

```
<alta>12/12/2020</alta>
</socio>
```

Queremos trasladar esta información a objetos Java que nos permitan procesarlos. Esta operación se llama *unmarshalling* (algo así como «desagrupamiento» en español, aunque también se le suele llamar «leer» un documento XML), que consiste en separar las distintas partes de la estructura de árbol de XML, propia del modelo DOM, donde los elementos de información están anidados unos dentro de otros, y todos ellos en el elemento raíz (en nuestro caso, *socio*), y convertirlos en objetos Java de distintas clases.

En primer lugar, tenemos que definir las clases necesarias. Dado que nuestro ejemplo es muy simple, lo haremos a partir del propio documento XML, aunque la forma correcta sería a partir del esquema.

En este caso solo necesitamos una clase, pero luego veremos un ejemplo más complejo. Es lógico suponer que el elemento raíz de nuestro documento XML se corresponda con una clase *Socio* y los elementos hijo *nombre*, *dirección* y *alta*, así como el atributo *id*, sean atributos de ella. Vamos a definir la clase *Socio* con los atributos que queremos que tenga e iremos viendo cómo añadir indicaciones (llamadas *anotaciones*), con información adicional para las conversiones entre XML y Java, que se realizarán en tiempo de ejecución. Todas las anotaciones empiezan por *@Xml* y se hallan en el paquete *javax.xml.bind.annotation*, que hay que importar. Se coloca cada una de ellas justo encima de la clase, atributo o propiedad a que se refieren. Por ejemplo, antes de la declaración del atributo *nombreSocio* de la clase *Socio* se escribe la anotación: *@XmlElement(name="nombre")*, que significa que ese atributo se corresponderá con un elemento simple, llamado '*nombre*', en el fichero XML.

La clase *Socio* sin anotaciones sería:

```
import javax.xml.bind.annotation.*;

public class Socio {
    private Integer identificacion;
    private String nombreSocio;
    private String dirección;
    private String fechaAlta;

    /*el constructor por defecto es obligatorio, aunque nosotros añadiremos otro
    que nos resultará útil: */
    public Socio() {
    }
    public Socio(Integer identificacion, String nombreSocio, String direccion,
    String fechaAlta) {
        this.identificacion = identificacion;
        this.nombreSocio = nombreSocio;
        this.direccion = direccion;
        this.fechaAlta = fechaAlta;
    }
    /*_resto de la implementación, incluyendo los getter, los setter y toString()
    para ver los resultados*/
}
```

Ahora vamos a añadir las anotaciones:

Justo encima de la definición de la clase irán las anotaciones que le atañen, que serán tres:

```
@XmlElement(name="socio") //el elemento raíz se llamará 'socio'
@XmlType(propOrder = {"nombreSocio", "direccion", "fechaAlta"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Socio {
    ...
}
```

La primera significa que la propia clase se corresponderá, en el documento XML, con un elemento raíz llamado `socio`, con elementos hijo o atributos. Todas las clases y tipos enumerados llevan una anotación `@XmlElement`. La segunda anotación establece el orden en que aparecerán los elementos hijo del elemento `socio` en el archivo XML (obsérvese que aquí deben aparecer los nombres que tienen los atributos en la clase `Socio`, no los que tendrán en el archivo XML). La tercera anotación establece que los elementos hijo del elemento raíz `socio` se tomarán automáticamente de los atributos (*fields* en inglés) no estáticos de la clase, aunque sean privados, salvo los que declaremos como transitorios (veremos un ejemplo más adelante). Otra opción sería tomarlos de las propiedades (los *getter* o los *setter*) con el valor `PROPERTY` en vez de `FIELD` o de los miembros públicos, ya sean atributos o propiedades, con `PUBLIC_MEMBER`.

A continuación, escribiremos anotaciones encima de las declaraciones de los atributos de la clase para especificar cómo se trasladarán al documento XML. Todos ellos serán elementos hijo o atributos del elemento raíz. El primero será:

```
@XmlAttribute(name = "id", required = true)
private Integer identificacion;
```

Esta anotación significa que el atributo Java `identificacion` se corresponderá, en el documento XML, con un atributo `id` del elemento raíz `socio`. Además, será obligatorio. Cuando una anotación recoge más de una opción dentro de sus paréntesis, van separadas por comas.

El siguiente atributo Java será:

```
@XmlElement(name = "nombre")
private String nombreSocio;
```

La anotación `@XmlElement` indica que el atributo `nombreSocio` se convertirá, en el documento XML, en un elemento simple con nombre `nombre` (hijo de `socio`).

Al atributo `direccion` no le pondremos ninguna anotación. Esto hará que se corresponda con un elemento XML del mismo nombre:

```
private String direccion;
```

La última anotación indicará que el atributo `fechaAlta` se va a corresponder con el elemento de nombre «alta».

```
@XmlElement(name="alta")
private String fechaAlta;
```

JAXB hace corresponder el tipo *date* de XML con el tipo obsoleto `XMLGregorianCalendar` de Java. Hay métodos que convierten este último en `LocalDate`, pero nosotros trabajaremos con cadenas, que se pueden pasar a `LocalDate`, cuando sea necesario, con el formateador correspondiente.

La clase `Socio` completa, con anotaciones, quedará:

```
import javax.xml.bind.annotation.*;

@XmlRootElement(name="socio") //el elemento raíz se llamará 'socio'
@XmlType(propOrder = {"nombreSocio","direccion","fechaAlta"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Socio {
    @XmlAttribute(required=true)
    private Integer id;
    @XmlElement(name="nombre")
    private String nombreSocio;
    private String dirección;
    @XmlElement(name="alta")
    private String fechaAlta;
    public Socio() {
    }
    public Socio(Integer id, String nombreSocio, String direccion,
String fechaAlta) {
        this.id = id;
        this.nombreSocio = nombreSocio;
        this.direccion = direccion;
        this.fechaAlta = fechaAlta;
    }
    /*_resto de la implementación, incluyendo los getter, los setter y toString()
para ver los resultados*/
}
```

Una vez construida la clase `Socio`, con todas las anotaciones referentes a la estructura del documento XML asociado, vamos a proceder a extraer la información de nuestro documento inicial, con los datos de *Martin Fisher*, y a crear con ellos el objeto `Socio` correspondiente.

El punto de entrada a la API JAXB para cualquier proceso de agrupamiento, desagrupamiento o validación es un objeto de la clase `JAXBContext`, que contiene toda la información necesaria. Para obtener una instancia de esta clase (un contexto), no disponemos de constructor, sino del método estático `newInstance()`, al que se le pasa como parámetro la clase raíz principal (en problemas más complejos habrá más de un elemento raíz) de nuestro documento.

En el programa principal, escribiremos:

```
JAXBContext contexto = JAXBContext.newInstance(Socio.class);
```

A partir del contexto, podemos crear un objeto `Marshaller` para agrupar (escribir en un archivo XML) o `Unmarshaller` para desagrupar (leer del documento XML y crear objetos `Socio`). En nuestro caso, queremos hacer lo segundo.

```
Unmarshaller um = contexto.createUnmarshaller();
```


Ahora procedemos a leer del archivo `socio.xml`, desagrupar sus distintos elementos y construir un objeto `Socio`, todo ello con la sentencia,

```
Socio s = (Socio) um.unmarshal(new File("socio.xml"));
```

El cast es necesario porque el método `unmarshal()` devuelve un objeto `Object`.

El nombre del archivo deberá llevar su ruta de acceso si no está en la carpeta raíz de nuestro proyecto.

Si hemos implementado el método `toString()` en la clase `Socio`,

```
System.out.println(s);
```

muestra por pantalla

```
Socio{id=23, nombre=Martin Fisher, direccion=43 Bass St, alta=12/12/2020}
```

Para hacer una agrupación a partir de una clase, es decir, escribir un documento XML a partir de un conjunto de datos Java, también tenemos que crear el contexto apropiado. Nosotros vamos a hacerlo con la misma clase `Socio`. Por tanto, nos sirve el mismo contexto que acabamos de usar. La diferencia es que ahora, a partir de él, tenemos que obtener un objeto `Marshaller` e invocar el método `marshal()`:

```
Marshaller m = contexto.createMarshaller();
```

Con este `Marshaller`, vamos a agrupar el socio,

```
Socio s1 = new Socio(1, "Armando Fuentes", "C/Fontanería 1", "01/09/1990");
```

Llamamos al método `marshal()` con el objeto Java que queremos escribir y el archivo de texto, con extensión XML, donde queremos que se escriba. Pero antes estableceremos una salida formateada con `setProperty()`. De lo contrario, se escribiría todo en una sola línea.

```
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);  
m.marshal(s1, new FileWriter("sociol.xml"));
```

Después de esta sentencia se creará, en la carpeta raíz del proyecto, el archivo de texto `socio1.xml` con el código XML correspondiente al objeto `s1`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<socio id="1">  
  <nombre>Armando Fuentes</nombre>  
  <direccion>C/Fontanería 1</direccion>  
  <alta>01/09/1990</alta>  
</socio>
```

Si solo queremos visualizar el resultado por pantalla sin guardarlo en un archivo, en vez de pasar al método `marshal()` un archivo de texto, le pasamos el flujo `System.out`, correspondiente al monitor.

```
m.marshal(s1, System.out);
```

Vamos a ver un caso algo más complejo, donde hace falta definir dos clases y, por tanto, dos elementos raíz. Además, tendremos que introducir una tabla. Partiremos del archivo *club.xml* con la información de un club y de sus socios.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<club>
  <nombre>Diogenes</nombre>
  <socios>
    <socio id="1">
      <nombre>Sherlock Holmes</nombre>
      <direccion>221B Baker St</direccion>
      <alta>12/12/1890</alta>
    </socio>
    <socio id="51">
      <nombre>Winston Churchill</nombre>
      <direccion>10 Downing St</direccion>
      <alta>13/02/1942</alta>
    </socio>
  </socios>
</club>
```

Como puede verse, aquí hay elementos de dos tipos, que tendrán que corresponderse con sendas clases: el club y los socios. La clase `Socio` ya la hemos definido y se corresponde exactamente con los socios de este archivo. La clase `Club` deberá tener un atributo `nombre` con el nombre del club y otro con la lista de los socios, que nosotros implementaremos por medio de una tabla de tipo `Socio`. En la práctica, estas listas de elementos se suelen implementar con colecciones, que nosotros no veremos hasta la Unidad 12, pero, de todas formas, las anotaciones son las mismas. Por otra parte, vamos a añadir un elemento que no queremos que se refleje en el archivo XML. Supongamos que, en nuestra aplicación Java, queremos gestionar el NIF del club, pero sin que aparezca en los archivos XML generados. En este caso, deberá existir un atributo `nif` en la clase `Club`, pero ningún elemento ni atributo `nif` en el archivo XML.

La clase `Club` tendrá la siguiente forma:

```
import java.util.Arrays;
import javax.xml.bind.annotation.*;

@XmlRootElement(name = "club")
@XmlType(propOrder = {"nombreClub", "listaSocios"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Club {
    @XmlElement(name = "nombre")
    private String nombreClub;
    @XmlElementWrapper(name = "socios")
    @XmlElement(name = "socio")
    private Socio[] listaSocios;
    @XmlTransient
    private String nif;
    public Club() {
```

```

    }
    public Club(String nombreClub, String nif) {
        this.nombreClub = nombreClub;
        this.listaSocios = new Socio[0];
        this.nif = nif;
    }
    public void nuevoSocio(Socio nuevo) {
        listaSocios = Arrays.copyOf(listaSocios, listaSocios.length + 1);
        listaSocios[listaSocios.length - 1] = nuevo;
    }
    //..resto de la implementación con getters, setters, etc
}

```

Lo primero que llama la atención son las anotaciones de la declaración de `listaSocios`, que es una tabla de elementos `Socio`.

```

@XmlElementWrapper(name = "socios")
@XmlElement(name = "socio")
private Socio[] listaSocios;

```

La primera anotación declara que `listaSocios` va a corresponderse con un elemento **envoltorio** (*wrapper*), de nombre «socios», con otros elementos en su interior. La segunda nos dice que esos otros elementos (correspondientes a objetos `Socio`) van a aparecer con el nombre «socio». Los elementos envoltorio son complejos, pero, a diferencia de lo que ocurre con los elementos raíz que se asocian con clases, sus elementos hijo son todos iguales, ya que proceden de tablas (o colecciones), cuyos elementos son también homogéneos.

Otra anotación nueva para nosotros es la que precede al atributo `nif`.

```

@XmlTransient
private String nif;

```

Significa que, al generar el fichero XML, no se reflejará en él el atributo `nif`. Por eso tampoco aparece en la lista de atributos de la anotación `@XmlType` del elemento raíz.

La forma de agrupar y desagrupar la clase `Club` es igual que la de `Socio`. Basta con crear el contexto con la clase `Club`. Java rastrea todas las relaciones con la clase `Socio`, que quedará incorporada al contexto.

```

JAXBContext contexto = JAXBContext.newInstance(Club.class);

```

Para desagrupar, creamos un `Unmarshaller` a partir del contexto y llamamos al método `unmarshall()` con el fichero XML como parámetro.

```

Unmarshaller um = contexto.createUnmarshaller();
Club c = (Club) um.unmarshal(new File("club.xml"));
System.out.println(c);

```

Se observa que el atributo `nif` de `c` tiene valor `null`, como cabía esperar, ya que no figura en documento `club.xml`.

Para agrupar, creamos un club con un par de socios.

```

Club c = new Club("Nautico", "1234");
Socio s1 = new Socio(1, "Juan Vela", "C/Galera 4", "03/02/2001");
Socio s2 = new Socio(2, "Amanda Lagos", "C/Siroco 21", "14/07/2002");
c.nuevoSocio(s1);
c.nuevoSocio(s2);

```

Con el mismo contexto, creamos el `Marshaller` y escribimos el resultado en el archivo `club2.xml`.

```

Marshaller m = contexto.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(c, new FileWriter("club2.xml"));

```

El archivo quedará así:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<club>
  <nombre>Nautico</nombre>
  <socios>
    <socio id="1">
      <nombre>Juan Vela</nombre>
      <direccion>C/Galera 4</direccion>
      <alta>03/02/2001</alta>
    </socio>
    <socio id="2">
      <nombre>Amanda Lagos</nombre>
      <direccion>C/Siroco 21</direccion>
      <alta>14/07/2002</alta>
    </socio>
  </socios>
</club>

```

Como puede verse, no aparece el `nif` del club, que sabemos tiene el valor «1234».