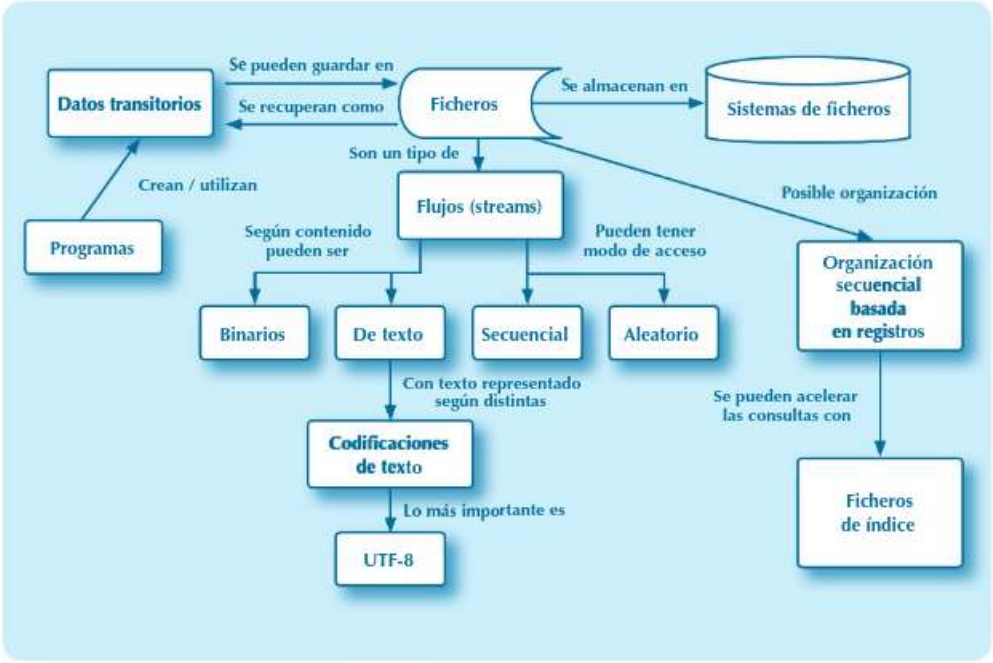
- Existen diversos sistemas de persistencia de datos que permiten a los programas de aplicación grabar datos transitorios como datos persistentes, y recuperar datos persistentes en memoria principal como datos transitorios.
- Los sistemas de persistencia de datos suelen funcionar en servidores que proporcionan servicios a las aplicaciones a través de protocolos estándares de red. Las aplicaciones suelen utilizar estos servicios a través de API (interfaces de programación de aplicaciones).
- Hay distintos tipos de sistemas de persistencia de datos. Entre ellos los basados en ficheros, las bases de datos relacionales y objeto-relacionales, las bases de datos de objetos, las bases de datos de XML y de otros tipos, denominadas en conjunto bases de datos NoSQL. Las bases de datos de XML se suelen considerar con frecuencia como NoSQL.
- Cada tipo de sistema proporciona una estructura básica para el almacenamiento de datos: las bases de datos relacionales, tablas; las bases de datos de XML, documentos XML con estructura jerárquica (en forma de árbol); las bases de datos de objetos, objetos complejos, con referencias a otros objetos y a colecciones de objetos; las diversas bases de datos NoSQL, diversas estructuras; y los ficheros son simples secuencias de *bytes*.
- Las API que permiten a los programas de aplicación acceder a datos persistentes suelen proporcionar iteradores para obtener uno a uno los resultados de las consultas.
- Los sistemas de persistencia de datos suelen proporcionar control y sincronización de accesos concurrentes y transacciones. Las bases de datos relacionales suelen tener un excelente soporte para transacciones. Las bases de datos NoSQL –sin incluir entre ellas las bases de datos de objetos ni las de XML– no suelen ofrecer soporte para transacciones porque prima la disponibilidad sobre la consistencia e integridad de los datos.
- Las bases de datos relacionales son, con diferencia, las más utilizadas. Están basadas en un sólido modelo formal: el modelo relacional. Se manejan mediante SQL, un lenguaje estándar declarativo y de muy alto nivel. Suelen utilizarse incluso para almacenar objetos y documentos de XML, introducidos en sucesivas revisiones del estándar SQL.
- El almacenamiento de objetos en bases de datos relacionales plantea una serie de problemas, conocidos en conjunto como *desfase objeto-relacional*. Pero existen técnicas y herramientas de ORM o correspondencia objeto-relacional que lo hacen posible.
- Las bases de datos de objetos permiten almacenar directamente objetos complejos que contienen referencias a otros objetos y a colecciones de objetos. Muchas se basan en o dan soporte al estándar ODMG 3.0, que incluye los lenguajes ODL para definición de datos y OQL para consulta de datos.
- Las bases de datos de XML nativas suelen organizar los documentos en una jerarquía de colecciones, en la que cada una puede contener documentos XML y de otros tipos. Estas bases de datos proporcionan, además, soporte para lenguajes estándares del W3C, como XQuery, XML Schema y XSL, y para API estándares tales como XML:DB y XQJ.

Mapa conceptual



Glosario

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

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

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

Fichero de texto. Fichero que contiene texto.

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

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

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

TRABAJO CON FICHEROS

- Clasificación según **Contenido**:
  - Ficheros Texto
  - Ficheros Binarios

Un fichero es simplemente una secuencia de *bytes,* con lo que en principio puede almacenar cualquier tipo de información (véase figura 1.3).

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

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

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

2.4. La clase File de Java

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

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

Métodos de la clase `File`

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

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

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

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

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

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

EXCEPCIONES

- Gestión Excepciones Java

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

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

```
// Excepción no gestionada durante la ejecución de un programa.
package excepcionesdivporcero;
public class ExcepcionesDivPorCero {
    public int divide(int a, int b) {
        return a/b;
    }
    public static void main(String[] args) {
        int a,b;
        a=5; b=2; System.out.println(a+"/"+b+"="+a/b);
        b=0; System.out.println(a+"/"+b+"="+a/b);
        b=3; System.out.println(a+"/"+b+"="+a/b);
    }
}
```

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

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

Captura y Gestión Excepciones

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

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

```java
// Excepción gestionada durante la ejecución de un programa.
package excepcionesdivporcerogest;
public class ExcepcionesDivPorCeroGest {
    public int divide(int a, int b) {
        return a / b;
    }
    public static void main(String[] args) {
        int a, b;
        a = 5; b = 2;
        try {
            System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 0; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
        try {
            b = 3; System.out.println(a + "/" + b + "=" + a / b);
        } catch (ArithmeticException e) {
            System.err.println("Error al dividir: " + a + "/" + b);
        }
    }
}
```

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

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

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

Métodos de la clase `Exception`

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

- Gestión Diferenciada Distintos Tipos Excepciones

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

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

```
// Gestion diferenciada de distintos tipos de excepciones

package excepcionesdiversas;

public class ExcepcionesDiversas {

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

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

```
Segunda cifra de 5*nums[0][0]/0: ERROR: aritmético 5*0/0
Segunda cifra de 5*nums[0][1]/1: ERROR: de otro tipo al calcular segunda cifra de: 5*1/1

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

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

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

- Declaración Excepciones Lanzadas por Método de Clase

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

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

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

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

public class ExcepcionesConThrows {

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

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

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

```
Inicialización y asignación de recursos
Cuerpo
Finalización y liberación de recursos
```

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

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

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

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

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

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

```
Inicialización y asignación de recursos
try (T1 r1=new T1(); T2 r2=new T2()) {
    // T1, T2 implementan Closeable o AutoCloseable

  Cuerpo

} catch(Excepcion_Tipo_1 e1) {
  Gestión de excepción de tipo 1
} catch(Excepcion_Tipo_2 e2) {
  Gestión de excepción de tipo 2
} catch(Exception e) {
  Gestión del resto de tipos de excepciones
}
finally {
Finalización y liberación de recursos
    // No es necesario r1.close()
    // No es necesario r2.close()
}
```

TRABAJO CON FICHEROS (Continuación)

- Formas Acceso Ficheros
  - ✓ Ficheros **Secuenciales**
  - ✓ Ficheros **Directos o Aleatorios**

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

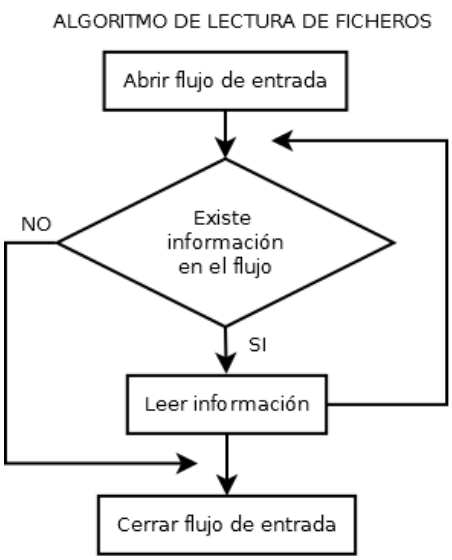
❖ Operaciones sobre Ficheros Java

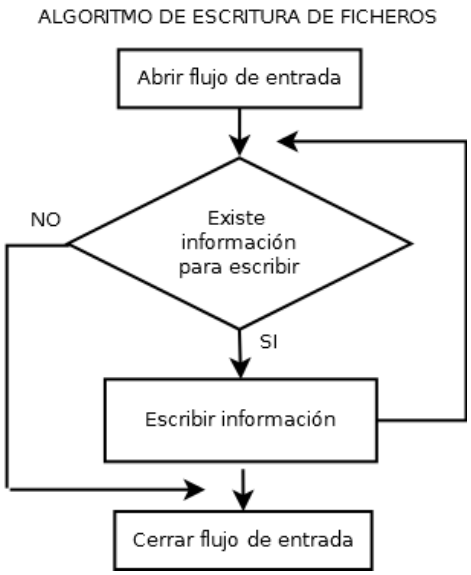
Independientemente del tipo de fichero (binario o de texto) y del tipo de acceso (secuencial o aleatorio), las operaciones básicas sobre ficheros son en lo esencial iguales, y se explicarán en este apartado. En los apartados siguientes se introducirán las particularidades de las clases que proporciona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

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

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

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





❖ Operaciones sobre Ficheros **Abiertos**

Normalmente las operaciones típicas que se realizan sobre un fichero una vez abierto son las siguientes:

- **Altas**:
  consiste en añadir un nuevo registro al fichero.
- **Bajas:**
  consiste en eliminar del fichero un registro existente.
  La eliminación puede ser
  - lógica, cambiando el valor de algún campo del registro que usemos para controlar dicha situación; o bien,
  - física, eliminando físicamente el registro del fichero. El borrado físico consiste muchas veces en reescribir de nuevo el fichero en otro fichero sin los datos que se desean eliminar y luego renombrarlo al fichero original.
- **Modificaciones:**
  consiste en cambiar parte del contenido de un registro.
  Antes de realizar la modificación será necesario localizar el registro a modificar dentro del fichero; y una vez localizado se realizarán los cambios y se reescribe el registro.
- **Consultas:**
  consiste en buscar en el fichero un registro determinado.

❖ Lectura y Escritura de Flujos

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

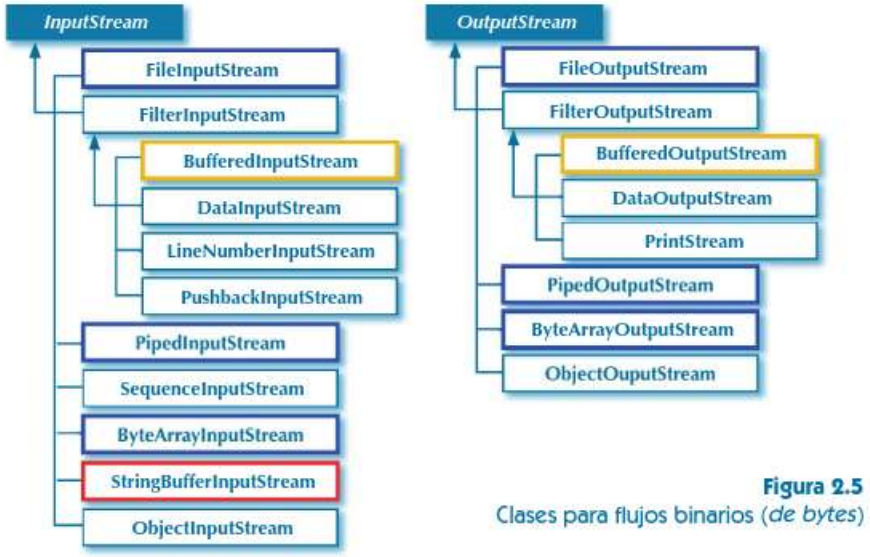


Figura 2.5
Clases para flujos binarios (*de bytes*)

Imagen Clases Flujos **Binarios**

TOMA NOTA

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

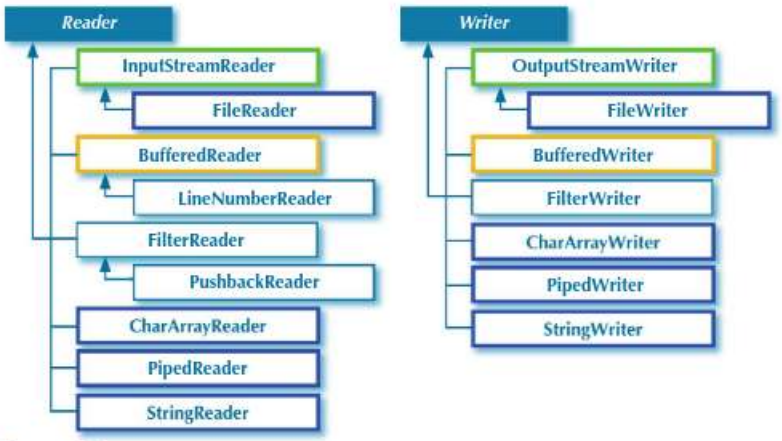


Figura 2.6
Clases para flujos de texto

Imagen Clases Flujos **Texto**

Clases básicas para entrada y salida a flujos

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

Figura 2.7. Lectura y escritura en ficheros binarios

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

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

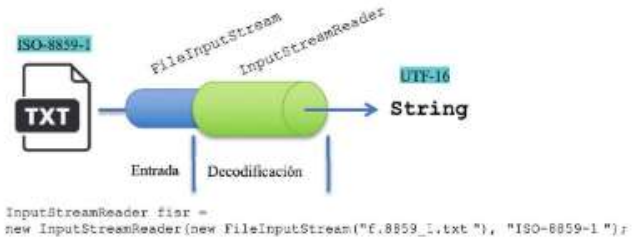


Figura 2.9. Lectura desde ficheros de texto con codificación distinta a la codificación por defecto

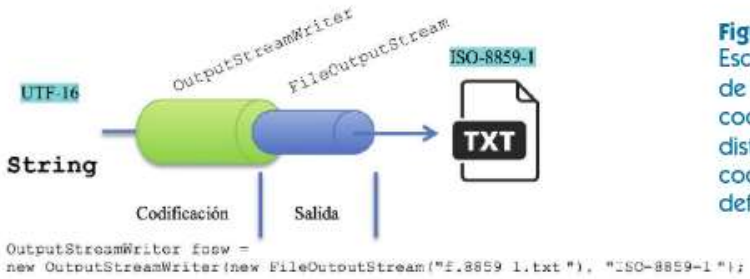


Figura 2.10. Escritura a ficheros de texto con codificación distinta a la codificación por defecto

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

TOMA NOTA

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

CUADRO 2.4
Clases que proporcionan buffering para entrada y salida a flujos

	Lectura	Escritura
Flujo binario	`BufferedInputStream`	`BufferedOutputStream`
Flujo de texto	`BufferedReader`	`BufferedWriter`

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

CUADRO 2.5
Flujo sin *buffering* y con *buffering*

Flujo sin buffering	Flujo con buffering
`new FileInputStream("f.bin")`	`new BufferedInputStream(new FileInputStream("f.bin"))`
`new FileOutputStream("f.bin")`	`new BufferedOutputStream(new FileOutputStream("f.bin"))`

CUADRO 2.5 (CON

`new FileReader("f.txt")`	`new BufferedReader(new FileReader("f.txt"))`
`new FileWriter("f.txt")`	`new BufferedWriter(new FileWriter("f.txt"))`

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

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

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

InputStream	Reader

`int read()`
`int read(byte[] buffer)`
`int read(byte[] buffer, int offset, int longitud)`
	`int read()`
	`int read(char[] buffer)`
	`int read(char[] buffer, int offset, int longitud)`
	`int read(CharBuffer buffer)`

`long skip(long n)`	`long skip(long n)`
	BufferedReader
	`String readLine()`

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

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

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

public class EscribeConNumeroDeLineas {

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

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

Ejercicios **Ficheros 1**

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

```
// Volcado hexadecimal de un fichero con FileInputStream

package volcadobinario;

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

public class VolcadoBinario {

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

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

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

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


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

Métodos para escritura de clases para gestión de flujos de salida

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

CUADRO 2.8 (CONT.

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

El siguiente programa escribe un texto en un fichero. Después lo cierra y lo vuelve a abrir en modo *append* para añadir nuevos contenidos al final. A menos que el fichero ya exista, en cuyo caso no hace nada. Si se hicieran estas mismas operaciones sobre un fichero existente, se perderían los contenidos del fichero. Se añaden saltos de línea con `newLine()`.

```
// Añadir contenidos al final de un fichero de texto

package escribeenflujosalida;

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

public class EscribeEnFlujoSalida {

    public static void main(String[] args) {

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

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

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

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

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

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

package arreglaficherotexto;

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

public class ArreglaFicheroTexto {

   public static void main(String[] args) {

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

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

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

FICHEROS ALEATORIOS

Los archivos de acceso secuencial son útiles para la mayoría de las aplicaciones, pero a veces son necesarios archivos de acceso aleatorio que permiten acceder a su contenido de forma secuencial o aleatoria.

La clase **`RandomAccessFile`** del paquete java.io implementa un archivo de acceso aleatorio. Puede ser usada tanto para la lectura como para la escritura de bytes. Dicha clase dispone de métodos para acceder al contenido de un fichero binario de forma aleatoria y para posicionarnos en una posición determinada del mismo. Esta clase no es parte de la jerarquía **InputStream** /**OutputStream**, ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Cuando queramos abrir un fichero de acceso aleatorio tendremos que crear un objeto de tipo **`RandomAccessFile`** y en el constructor indicaremos la ruta del fichero y el modo de apertura: sólo lectura "**`r`**", o lectura/escritura "**`r/w`**".

Hay dos posibilidades para abrir un fichero de acceso aleatorio:

```
RandomAccessFile(String path, String modo); // Con el
nombre del fichero
RandomAccessFile(File fichero, String modo); // Con un
objeto File
```

Ejemplos:

```
RandomAccessFile fichero = new RandowAccessFile("Datos.txt",
"rw");
RandomAccessFile fichero = new RandowAccessFile(File archivo,
"rw");
```

Todo objeto, instancia de **`RandomAccessFile`** soporta el concepto de puntero que indica la posición actual dentro del archivo. Cuando en el fichero se crea el puntero se coloca en 0, apuntando al principio del mismo. Las sucesivas llamadas a los métodos read() y write() ajustan el puntero según la cantidad de bytes leídos o escritos.

Desplazamiento: cualquier operación de lectura/escritura de datos se realiza a partir de la posición actual del **"puntero"** del archivo.

Principales métodos de la clase `RandomAccessFile`

Método	Funcionalidad
`RandomAccessFile(File file, String mode)` `RandomAccessFile(String name, String mode)`	Constructor. Abre el fichero en el modo indicado, si se dispone de permisos suficientes. *r:* modo de solo lectura. *rw:* modo de lectura y escritura. *rwd, rws:* Como *rw* pero con escritura síncrona. Esto significa que todas las operaciones de escritura (de datos con *rwd* y de datos y metadatos con *rws*) deben haberse completado cuando termina la función. Esto puede hacer que la llamada a la función tarde más, pero asegura que no se pierde información crítica ante una caída del sistema.
`void close()`	Cierra el fichero. Es conveniente hacerlo siempre al final.
`void seek(long pos)`	Posiciona el puntero en la posición indicada.
`int skipBytes(int n)`	Intenta avanzar el puntero el número de *bytes* indicado. Se devuelve el número de *bytes* que se ha avanzado. Podría ser menor que el solicitado, si se alcanza el fin del fichero.

`int read()`
`int read(byte[] buffer)`
`int read(byte[] buffer, int offset, int longitud)`
	Lee del fichero. Según la variante, un *byte* o hasta llenar el *buffer,* o el número de *bytes* indicados, que se copiarán en la posición indicada *(offset)* del *buffer*. Devuelve el número de *bytes* leídos, o -1 si no se pudo leer nada porque el puntero estaba al final del fichero.

`void readFully(byte[] buffer)` `readFully(byte[] buffer, int offset, int longitud)`	Como `int read(byte[] b)`, pero si no se puede leer hasta llenar el *buffer,* o el número de *bytes* indicado, porque se llega al final del fichero, se lanza la excepción `IOException`. Útil cuando se sabe que se podrá leer hasta llenar el *buffer* o el número de *bytes* indicados. En cualquier caso, la eventualidad de que no se pueda completar la lectura se puede gestionar capturando la excepción.
`String readLine()`	Lee hasta el final de la línea de texto actual.

`void write(int b)`
`void write(byte[] buffer)`
`void write(byte[] buffer, int offset, int longitud)`
	Escribe en el fichero. Según la variante, un *byte,* o todos los contenidos del *buffer,* o el número de *bytes* indicados a partir de la posición indicada *(offset)*. Si alcanza el fin del fichero, siguen escribiendo.

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

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

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

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

package ficheroaccesoaleatorio;

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

public class FicheroAccesoAleatorio {

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

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

    public long getNumReg() {
        return numReg;
    }

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

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

     }

        public static void main(String[] args) {
            List campos = new ArrayList();
            campos.add(new Pair("DNI", 9));
            campos.add(new Pair("NOMBRE", 32));
            campos.add(new Pair("CP", 5));
            try {
            FicheroAccesoAleatorio faa = new FicheroAccesoAleatorio("fic_acceso_
               aleat.dat", campos);
            Map reg = new HashMap();
            reg.put("DNI", "56789012B");
            reg.put("NOMBRE", "SAMPER");
            reg.put("CP", "29730");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "89012345E");
            reg.put("NOMBRE", "ROJAS");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "23456789D");
            reg.put("NOMBRE", "DORCE");
            reg.put("CP", "13700");
            faa.insertar(reg);
            reg.clear();
            reg.put("DNI", "78901234X");
            reg.put("NOMBRE", "NADALES");
            reg.put("CP", "44126");
            faa.insertar(reg,1);
        } catch (IOException e) {
            System.err.println("Error de E/S: " + e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Ejercicios **Ficheros 3**

❖ Organización de Ficheros

✓ Oganización Secuencial

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

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

- Las búsquedas son muy ineficientes. Para buscar cualquier registro, es necesario recorrer secuencialmente el fichero desde el principio hasta que se encuentre el registro que se busca o hasta llegar al final del fichero sin encontrarlo. Ordenar los registros evitaría tener que recorrer todos los ficheros cuando no está el registro que se busca. Pero ordenar por DNI no servirá de nada si se busca por nombre, por ejemplo. Además, mantener el fichero siempre ordenado complica las operaciones de inserción, borrado o modificación de registros (esto último en el caso en que se modifique el campo por el que está ordenado el fichero).
- El borrado de un registro es muy ineficiente porque obliga a correr una posición hacia atrás todos los registros siguientes. Una posible mejora podría ser marcar el registro como borrado mediante alguna marca especial. Por ejemplo, un carácter determinado en la primera posición, o dejar en blanco los campos que forman la clave, en este caso el DNI.
- La inserción de registros sería muy eficiente. Bastaría añadir el nuevo registro al final. A no ser, como ya se ha visto, que el fichero estuviera ordenado.

✓ Oganización Secuencial Indexada

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

Un fichero de índice no es más que un fichero secuencial ordenado cuyos registros contienen dos campos: uno para un valor del campo para el que se crea el índice, y otro que indica la posición en el fichero secuencial (véase figura 1.5).

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

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

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

OBJETOS SERIALIZABLES

Hemos estudiado como se guardan los tipos primitivos en un fichero, pero por ejemplo, tenemos un objeto de tipo empleado con varios atributos (nombre, dirección, salario, departamento, etc.) y queremos guardarlo en un fichero tendríamos que guardar cada atributo que forma parte del objeto por separado, esto es engorroso si tenemos varios objetos. Por ello Java nos permite guardar objetos en ficheros binarios para poder hacerlo, el objeto tiene que implementar la interfaz *Serializable* que dispone de una serie de métodos con los que podemos guardar y leer objetos en ficheros binarios.

- **Escritura de Objetos en un Fichero**

Para que un objeto pueda ser almacenado en disco, es necesario que la clase a la que pertenece sea **serializable**. Esta característica la poseen todas las clases que implementan la interfaz **`java.io.Serializable`**.

La **serialización** es el proceso por el cual un objeto o una colección de objetos se convierten en una serie de bytes que pueden ser almacenados en un fichero y recuperado posteriormente para su uso. Se dice que el objeto u objetos tienen **persistencia**. Cuando serializamos un objeto, almacenamos la estructura de la clase y todos los objetos referenciados por esta.

La interfaz **`Serializable`** no contiene ningún método, basta con que una clase la implemente para que sus objetos puedan ser serializados por la máquina virtual y por lo tanto almacenada en disco.

Ejemplo: Crea una clase Persona cuyos objetos encapsulan el nombre y la edad de una persona. Estos objetos pueden ser transferidos a disco, ya que Persona implementa la interfaz **`Serializable`**.

El primer paso es crear un objeto `FileOutputStream` que permita añadir información al fichero o sobreescribirla, para ello utilizamos los **constructores**:

```
FileOutputStream(File fichero, boolean append);
FileOutputStream(String path, boolean append);
```

Para escribir objetos en disco el segundo paso es crear un objeto de la clase `ObjectOutputStream`, cuyos constructores son:

```
ObjectOutputStream(); // Para crear un objeto que permite escribir objetos de java sobre cualquier dispositivo.
ObjectOutputStream(OutputStream); // Crea un flujo que permite escribir objetos de java en el objeto OutputStream que se pasa como argumento.
```

Una vez creado el objeto, la clase dispone de los mismos métodos que `DataOutputStream` más:

`void writeObject(Object)//` Para escribir el objeto en el disco. Propagan la excepción `IOException`.

- **Lectura de Objetos en un Fichero**

Cuando se recupera un objeto del disco mediante la llamada a `readObject()`, se produce la **deserialización** del objeto que consiste en la reconstrucción de éste a partir de la información recuperada.

Durante este proceso, los datos miembro no serializables (los que han sido heredados de una clase no serializable) serán inicializados utilizando el constructor por defecto de su clase.

Por el contrario, los datos miembro de la clase objeto serializado serán restaurados con los valores almacenados.

Para leer objetos de un fichero que han sido almacenados mediante `ObjectOutputStream`, hay que utilizar la clase `ObjectInputStream`.

El procedimiento es similar a los anteriores

El primer paso es crear un objeto `FileInputStream` que permita recuperar información del fichero, para ello utilizamos los **constructores**:

```
FileInputStream(File fichero);
FileInputStream (String path);
```

El segundo paso es a partir del objeto `FileInputStream` crear un objeto `ObjectInputStream` para realizar la escritura. El **constructor** es:

`ObjectInputStream(); //` Para crear un objeto que permite leer objetos de java sobre cualquier dispositivo.

`ObjectInputStream (InputStream); //` Crea un flujo que permite leer objetos de java en el objeto InputStream que se pasa como argumento.

La clase `OuputInputStream` proporciona métodos para leer datos desde un fichero:

Mismos métodos que `DataOutputStream` más:

`object readObject(); //` Para leer un objeto asociado al flujo y devolverlo de tipo `Object`. Hay que hacer obligatoriamente un cast a la clase a la que lo vamos a convertir.
Persona obj=(Persona) flujolectura.readObject();

Todos los métodos propagan una excepción de la clase `IOException`.

El método `readObject()` propaga la excepción `ClassNotFoundException`

(Cuando se intenta hacer un cast al objeto que se ha leído de una case que no es), y la excepción `StreamCorruptedException` (cuando se intenta hacer más de una operación de lectura a través de la clase `ObjectInputStream`) .

- ***Problemas con la clase ObjectOutputStream***

1º) Cuando escribimos un objeto, es como si guardara el array de bytes en el interior. Si cambiamos los valores de los atributos de ese objeto y volvemos a escribirlo el **ObjetOutputStream** lo escribe nuevamente, pero con los datos antiguos. Da la impresión de que no se entera del cambio y no recalcula los bytes que va a escribir en el fichero.

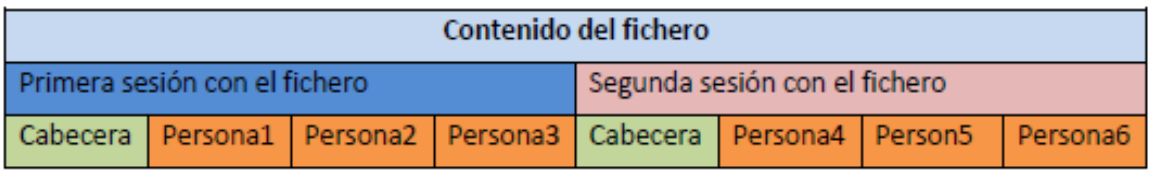
Esto puede evitarse de tres formas:

- Haciendo un **new** de cada objeto que queramos escribir, sin reaprovechar instancias.
  Esto es lo que se ha hecho en el código anterior.
- Usar el método **writeUnshared()** en vez de **writeObject()**. Este método funcionará bien si cambiamos **TODOS** los atributos de la clase **Persona**. Si no modificamos uno de los atributos, obtendremos resultados extraños en ese atributo.
- Llamando al método **reset()** de **ObjectOutputStream** después de escribir cada objeto. Aunque funciona, no parece demasiado ortodoxo.

2º) Un segundo problema es que al instanciar el objeto **ObjectOutputStream**, escribe unos bytes de cabecera en el fichero, antes incluso de que escribamos nada. Como el **ObjectInputStream** lee correctamente estos bytes de cabecera, aparentemente no pasa nada y ni siquiera nos enteramos que existen.
El problema se presenta si escribimos unos datos en el fichero y lo cerramos. Luego volvemos a abrirlo para añadir datos, creando un nuevo **ObjectOutputStream** así

```
/* El true del final indica que se abre el fichero para añadir
datos al final del fichero */
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero,true));
```

Esto escribe una nueva cabecera justo al final del fichero. Luego se irán añadiendo los objetos que vayamos escribiendo. El fichero contendrá lo del dibujo, con dos cabeceras.



Cuando leemos el fichero, lo primero será crear el **ObjectInputStream**, este leerá la cabecera del principio y luego se pone a leer objetos. Cuando llegamos a la segunda cabecera que se añadió al abrir por segunda vez el fichero para añadirle datos, obtendremos un error **StreamCorruptedException** y no podremos leer más objetos.
Una solución es evidente, no usar más que un solo **ObjectOuptutStream** para escribir todo el fichero. Sin embargo, esto no es siempre posible.
Una solución es hacernos nuestro propio **ObjectOutputStream**, heredando del original y redefiniendo el método **writeStreamHeader()**, vacío, para que no haga nada.

```
/* Redefinición del método de escribir la cabecera para que
no haga nada. */
protected void writeStreamHeader() throws IOException
{
}
```

TRABAJO CON FICHEROS XML

XML (*eXtensible Markup Languaje – Lenguaje de Etiquetado Extensible*)

- es un metalenguaje, es decir; un lenguaje para la definición de lenguajes de marcado. Nos permite jerarquizar y estructurar la información, así como describir los contenidos dentro del propio documento.

Los ficheros XML son ficheros de texto escritos en XML donde la información está organizada:

- de forma secuencial
- y en orden jerárquico.

Existen una serie de marcas especiales como son los símbolos menor que < y mayor que >, que se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre y puede tener 0 o más atributos.

Los ficheros XML se pueden utilizar:

- Para proporcionar datos a una base de datos o para almacenar copias de partes del contenido de la base de datos.
- Para escribir ficheros de configuración de programas
- En el protocolo SOAP (Simple Object Access Protocol), para ejecutar comandos en servidores remotos; la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o **parser**. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son:

  - **DOM**: *Modelo de Objetos de Documento*
  - y **SAX**: *API Simple para XML*.

• **DOM**: un procesador XML que utilice este planteamiento almacena la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos (de arriba abajo y también se puede volver atrás) y se analiza a qué tipo particular pertenecen. Podemos modificar cualquier nodo del árbol.

Tiene su origen en el W3C[1]. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.

Con ficheros XML pequeños no tendremos problemas, pero si tuviéramos un árbol muy muy grande entonces tendríamos una falta de **heap space**.

• **SAX**: un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta, etc.) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento prácticamente no consume memoria, pero por otra parte, impide tener una visión global del documento por el que navegar.
Al contrario que con DOM, al procesar en SAX no vamos a tener la representación completa del árbol en memoria, pues SAX funciona con eventos. Esto implica:

- Al no tener el árbol completo no puede volver atrás, pues va leyendo secuencialmente.
- La modificación de un nodo y la inserción de nuevos nodos son mucho más complejas.
- Como no tiene el árbol en memoria es mucho más **memory friendly**, de modo que es la única opción viable para casos de ficheros muy grandes, pero demasiado complejo para ficheros pequeños.

❖ DOM

Para poder trabajar con **DOM** en Java necesitamos las clases e interfaces que componen el paquete ***org.w3c.dom*** (contenido en el JSDK) y el paquete ***javax.xml.parsers*** del API estándar de Java que proporciona un par de clases abstractas que toda implementación **DOM** para Java debe extender. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos (fichero, InputStream, etc.) Contiene dos clases fundamentales:
DocumentBuilderFactory y ***DocumentBuilder***.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM.

Para eso usaremos el paquete ***javax.xml.transform***, que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos **DOM** entre otros.

Los programas Java que utilicen **DOM** necesitan interfaces, algunas son:

Interface	Función
Document	Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
Element	Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos
Node	Representa cualquier nodo del documento
NodeList	Contiene una lista con los nodos hijos de un nodo.
Att	Permite acceder a los atributos de un nodo
Text	Son los datos carácter de un elemento
CharacterData	Representa los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
DocumentType	Proporciona información contenida en la etiqueta <!DOCTYPE>

✓ Leer un documento XML con DOM

Para leer un documento XML, creamos una instancia de **DocumentBuilderFactory** para construir el parser y cargamos el documento con el método **parse().**

```
DocumentBuilder db = dbf.newDocumentBuilder();
Document documento = db.parse(new File("Personas.xml"));
```

Obtenemos la lista de nodos con nombre *personas* de todo el documento:

```
NodeList personas = documento.getElementsByTagName("persona");
```

Se realiza un bucle para recorrer la lista de nodos. Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función **getNodo().**

❖ SAX

SAX (API Simple para XML) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML. Permite analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como hace DOM), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. SAX es más complejo de programar que DOM, es un API totalmente escrita en Java e incluida dentro de JRE que nos permite crear nuestro propio parser XML.
La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin de documento (***startDocument()*** y ***endDocument()***), la etiqueta de inicio y fin de un elemento (***startElement()*** y ***endElement()***), los caracteres entre etiquetas (***characters()***), etc.

Ejemplo en Java en el que se muestran los pasos básicos necesarios para hacer que se puedan tratar los eventos.
En primer lugar se incluyen las clases e interfaces de SAX:

```
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
```

Se crea un objeto procesador de XML, es decir un ***XMLReader***[3], durante la creación de este
objeto se puede producir una excepción (***SAXException***) que es necesario capturar:

```
XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
```

A continuación, hay que indicar al ***XMLReader*** qué objetos poseen los métodos que tratarán

los eventos. Estos objetos serán normalmente implementaciones de las siguientes interfaces:

Interface	Función
ContentHandler	Recibe las notificaciones de los eventos que ocurren en el documento.
DTDHandler[4]	Recoge eventos relacionados con la DTD (Declaración de tipo de documento
ErrorHandler	Define métodos de tratamiento de errores
EntityResolver	Sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
DefaultHandler	Clase que provee una implementación por defecto para todos sus métodos, el programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML.

DefaultHandler es una clase que va a procesar cada evento que lance el procesador SAX.

Basta con heredar del manejador por defecto de **SAX *DefaultHandler*** y sobreescribir los métodos correspondientes a los eventos deseados. Los más comunes son:

- ***startDocument***: se produce al comenzar el procesado del documento xml.
- ***endDocument***: se produce al finalizar el procesado del documento xml.
- ***startElement***: se produce al comenzar el procesado de una etiqueta xml. Es aquí donde se leen los atributos de las etiquetas.
- ***endElement***: se produce al finalizar el procesado de una etiqueta xml.
- ***characters***: se produce al encontrar una cadena de texto.

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos ***XMLReader***:

setContentHandler(),
setDTDHandler(), setEntilyResolver(), setErrorHandler(); cada uno trata un tipo de evento y está asociado con una interfaz determinada.

```
GestionContenido gestor = new GestionContenido();
procesadorXML.setContentHandler(gestor);
```

A continuación se define el fichero XML que se va a leer mediante un objeto ***InputSource***:

```
InputSource ficheroXML = new InputSource("Personas.xml");
```

Por último, se procesa el documento XML mediante el método ***parse()*** del objeto XMLReader, le pasaremos un objeto InputSource:

```
procesadorXML.parse(ficheroXML);
```

Serialización de Objetos XML

Para serializar objetos Java a XMl o viceversa necesitamos la librería **XStream**. Para poder utilizarla debemos descargar los fichero JAR desde la web: http://xstream.codehaus.org/download.html, para el ejemplo descargaremos el fichero ***Binary distribution*** (*xstream-distribution-1.4-2-bin.zip*) lo descomprimimos y buscamos el fichero JAR ***xstream-1.4.7.jar*** que está en la carpeta ***lib.*** También necesitamos el fichero ***kxml2-2.3.0.jar*** que se puede descargar desde el apartado ***Optional Dependencies***.

Partimos del fichero *"Alumnos.Dat"* que contiene objetos *Alumno*. El proceso es crear una lista de objetos *Alumno* y la convertiremos en un fichero de datos XML. Necesitamos la clase *Alumno* y la clase *ListaAlumno* en la que definimos una lista de objetos *Alumno* que pasaremos al fichero XML.

Conversión de Ficheros XML a otro formato

XSL (*Extensible Stylesheet Language*) son recomendaciones del Word Wide Web Consortium (http://www.w3.org/Style/XSL/) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo **XSL** describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja puede contener elementos de reglas que representan a las reglas de construcción y elementos de reglas de estilo que representan a las reglas de mezclas de estilos.

En el ejemplo vamos a ver como a partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de esos datos se puede generar un fichero HTML usando el lenguaje Java.