Unidad 2

Ficheros

Acceso a datos

Índice

Introducción

En este tema podremos ver una gran cantidad de información relacionada con los ficheros, desde su naturaleza, empleo y organización.

En primer lugar, comenzaremos con el estudio de los distintos tipos de datos que podemos encontrar, así como el empleo que podemos hacer de cada uno de ellos. Junto con los datos podremos estudiar cómo estos forman ficheros y que tipo de ficheros podremos encontrar.

En segundo lugar, estudiaremos las excepciones que podemos realizar con java y los tipos de ellas, con el fin de que podamos realizar operaciones sin que se produzcan grandes errores.

En tercer lugar, estudiaremos las operaciones que podemos llevar a cabo con los ficheros, teniendo en cuenta la diferencia entre ficheros binarios y de texto. Terminaremos con la explicación de los dos posibles tipos de organización de los ficheros, aleatoria o indexada.

Al finalizar esta unidad

- Conoceremos las diferencias entre los ficheros binarios y los de texto, así como las diferencias que existen al llevar acciones sobre ellos, junto a esto encontraremos el estudio de los procesos de recodificación.
- Habremos estudiado la naturaleza de un buffer y la función que este puede realizar en los ficheros.
- Habremos descrito los diferentes datos, persistentes y transitorios o temporales, y dónde podemos encontrar cada uno de ellos.
- Sabremos distinguir los tipos de organización que podemos encontrar o realizar en los ficheros.

2.1. Persistencia de datos en ficheros

El fichero se emplea de una u otra forma, para el almacenamiento de información de todas las bases de datos. Es un elemento esencial, ya que forma la base de la pirámide en las bases de datos, siendo ellos los últimos contenedores de la información en bytes. En el empleo directo de ficheros como una base de datos se ha visto reducido en gran medida, ya que sus desventajas eran relevantes, pero aún se siguen empleando, generalmente como ficheros indexados en ciertas situaciones, como puede ser la banca o los sistemas de correo, donde este sistema de ficheros indexados sigue siendo muy útil.

A pesar de que aún se sigue empleando, para servicios específicos o muy sencillos, sus grandes desventajas le han llevado a perder mucho terreno ante sus competidores, las nuevas bases de datos como las relacionales o las NoSQL, que permiten mayores rendimientos en muchos sectores.

2.2. Tipos de ficheros según su contenido

Podemos denominar *fichero* una secuencia de bytes almacenada. Identificados por su nombre y ubicación en un directorio estos pueden contener cualquier tipo de información.

Podemos encontrar dos tipos de ficheros según su contenido:

- **Ficheros de texto:** almacena cadena de caracteres. Puede ser modificado por cualquier editor de texto como el blog de notas. Su acceso habitual es de forma secuencial. Si se quiere acceder al final del fichero hay que recorrer todo el fichero.
- **Ficheros binarios:** almacena bytes. Puede contener cualquier tipo de información son los empleados para almacenar programas, y es necesario un programa para poder visualizarlo. El acceso puede realizarse de forma secuencial o acceso directo.

Al bloque de datos que permanecen juntos al ser leídos o escritos se le denomina registro.

2.3. Codificaciones para texto

La codificación es imprescindible para cualquier tipo de almacenamiento. El almacenamiento se realiza en secuencias de bytes, pero tras codificarse su secuencia cambiará, esto se realiza con el fin de poseer un mismo tipo de secuencia de *bytes*.

En la actualidad Unicode y UTF-8 han permitido reducir el número de tipos de codificaciones, aunque aún se emplean otras, en especial en Windows.

Una de las características de UTF-8 es que es compatible con ASCII, lo que permite que se pueda visualizar ASCII correctamente en él, este ha sido una de las principales causas por las que UTF-8 ha tenido una gran aceptación.

La extensión de UTF-8 es tan amplia que se recomienda que todo se almacene en él, salvo que se recomiende por razones especiales. Debemos tener cuidado al introducir elementos de otros sistemas o programas, ya que la codificación puede no coincidir, en esos casos se debe de llevar a cabo un proceso de conversión llamado recodificación.

2.4. La clase File de Java

Podemos obtener información relativa a directorios y ficheros mediante el empleo de la clase File, igualmente con el podemos emplear diversas operaciones como renombrar, borrar, etc. Existen más métodos utilidad con los que podemos operar con sistemas de ficheros. En el siguiente cuadro podremos observar los métodos más comunes.

Métodos comunes de File	
Operación	**Descripción**
`boolean exists()`	Comprueba si la ruta existe.
`boolean isFile()`	Comprueba si es un fichero.
`boolean isDirectory()`	Comprueba si es un directorio.
`File[] listFiles()`	Obtiene un listado de tipo File de los ficheros que hay en la ruta.
`String getName()`	Obtiene el nombre del fichero o directorio.
`String getParent()`	Obtiene el directorio padre.
`String getPath()`	Obtiene la ruta.
`Boolean canRead()`	Comprueba si se puede leer.
`Boolean canWrite()`	Comprueba si se puede escribir.
`Boolean canExecute()`	Comprueba si se puede ejecutar.

2.5. Gestión de excepciones en Java

Una **excepción** es un evento ante una situación no prevista por un error funcional o lógico, unos parámetros con valores no definidos, un flujo incorrecto, una acción no prevista o cualquier problema externo de conectividad, hardware, etc. Que se produce en tiempo de ejecución. Por ejemplo, no se pueda establecer una conexión a una base de datos, no se pueda abrir un fichero, el programa se quede sin memoria, se tratarte un tipo de dato incorrecto, intentar acceder a una posición de un array inexistente, etc. Las excepciones son gestionadas en la mayoría de los lenguajes de programación, en concreto, java ya dispone de unas excepciones predefinidas que se explicará en los siguientes puntos.

Cuando se ejecuta un programa se empieza desde la clase que contiene el método *main* y se establece un flujo de llamadas a otras clases y métodos según la funcionalidad, denominado **pila de llamadas**. Cuando se produce una excepción en cualquier punto del código se propagará por toda la pila hasta llegar al método *main* y finalizar el programa.

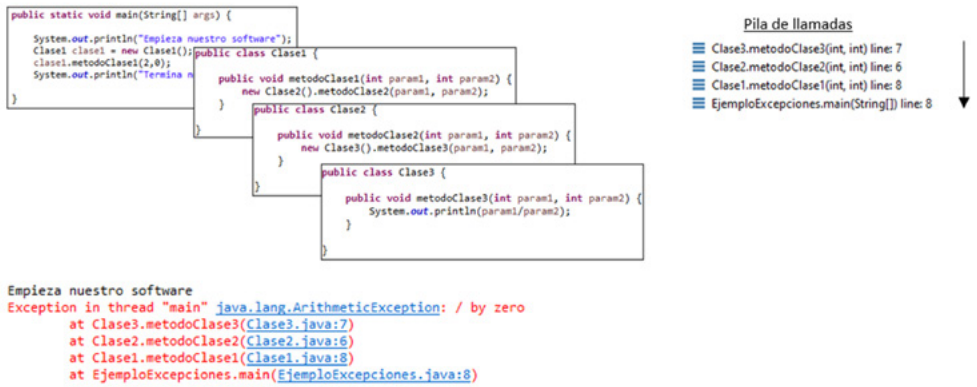

Imagen 1. Propagación de una excepción al dividir por cero

Para evitar que cada vez que se produzca una excepción, recorra toda la pila de llamadas y termine el programa, se realizará un **tratamiento de excepciones** para realizar acciones de corrección o alternativas para seguir con el correcto funcionamiento del programa.

Hacer un bueno desarrollo y control de excepciones permitirá tener un programa más robusto y evitará cualquier anomalía que haga que no funcione correctamente. Desde la etapa de codificación es recomendable tener un listado de todos los puntos donde puede producirse una excepción y diseñar su tratamiento si es necesario. Para ello es recomendable:

- Tener listado todas las excepciones posibles.
- Poner un identificador exclusivo de cada excepción.
- Saber el flujo de llamadas y como se propagará la excepción.
- Separar el código destinado a controlar errores del resto de código.

2.5.1. Captura y gestión de excepciones

Las excepciones se pueden tratar e incluso, en algunos casos, es una acción obligatoria, pero antes de hacer el tratamiento hay que determinar que parte de código puede producir una excepción, si es necesario realizar alguna validación previa o incluso provocar o invocar una excepción por iniciativa nuestra. A continuación, se detalla diferentes formas de manejar excepciones.

Se indicará que código va a producir una excepción, el tratamiento que se vaya a hacer si se produce y opcionalmente podemos indicar un tratamiento final se produzca o no la excepción. Estas capturas se realizarán con los bloques `try/catch/finally`.

Sintaxis:

```
try {
     // Código que puede producir una excepción.
} catch (Tipo excepcion) {
     // Código para dar tratamiento a la excepción.
} [catch (Tipo excepcion1)] {
     // ..
} finally {
     // Código que se realizará si se produce la excepción o no.
}
```

La parte del catch es posible definir más de un bloque para capturar diferentes excepciones del bloque try. El orden en que se especifique cada catch es importante según la jerarquía de excepciones. Si se indica como primera excepción la raíz todos los catch anteriores no se usarán nunca. Hay que especificar en orden de menor rango de la jerarquía a mayor.

Orden correcto de los Catch

```
try {
} catch (ArithmeticException e1) {
} catch (ArrayIndexOutOfBoundsException e2) {
} catch (Exception e) {
}
```

Orden incorrecto de los Catch

```
try {
} catch (Exception e) {
} catch (ArithmeticException e1) {
} catch (ArrayIndexOutOfBoundsException e2) {
}
```

Imagen 2. Uso de más de un bloque catch y su orden

2.5.2. Gestión diferenciada de distintos tipos de excepciones

Dentro de un bloque podemos gestionar por separado **try{}** y **catch {}** de modo que podamos incluir diferentes tipos de excepciones. Con el fin de que no queden sin gestionar, es recomendable añadir un manejador para excepción.

Con este tipo de entrada podremos diferenciar distintas excepciones con respuestas diferentes para cada una de las situaciones posibles.

```
try {

} catch (ArithmeticException | ArrayIndexOutOfBoundsException e ) {
    // Código para la excepciones Arithmetic y ArrayIndexOutOfBounds
} catch (Exception e) {
    // Código para el resto de excepciones de la clase padre de Exception
}
```

Imagen 3. Ejemplo de sintaxis para try{} y catch{}

2.5.3. Declaración de excepciones lanzadas por un método de clase

Las excepciones pueden ser numerosas, pero no debemos olvidar gestionar ninguna con catch () {}, pues esto supondrá un error en el programa. Con el fin de evitar estos errores deberemos ir incluyéndolos cuando aparezcan o emplear el modificador *throws*.

```
public static void main(String[] args) {
    try {
        // Calculadora con las operaciones de: 0 - Suma, 1 - Resta, 2- Producto, 3- División
        calculadora(3, 0, 0);
    } catch (ArithmeticException e) {
        System.out.println("Se ha producido el siguiente error: "+e.getMessage());
        System.out.println("En: "+e.getClass());
        for (StackTraceElement traceElement : e.getStackTrace())
            System.out.println("\tat " + traceElement);
    }
}

public static int calculadora(int operacion, int param, int param2)
        throws ArithmeticException{

    switch (operacion) {
        case 0:
                return param + param2;
        case 1:
                return param - param2;
        case 2:
                return param * param2;
        case 3:
                return param / param2;

    }

    return -0;
}
```

```
Se ha producido el siguiente error: / by zero
En: class java.lang.ArithmeticException
        at EjemploPropagacion.calculadora(EjemploPropagacion.java:30)
        at EjemploPropagacion.main(EjemploPropagacion.java:8)
```

Imagen 4. Ejemplo de propagación de excepciones

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

Los bloques de Java se suelen estructurar en tres fases:

- Inicio y asignación de recursos.
- Cuerpo.
- Finalización y liberación de recursos

La primera y última parte deben ejecutarse siempre, de modo que se forme un círculo.

```
for{
      try { cuerpo }
          catch (Excepción 1)
              {Gestión de la excepción 1}
          catch (Excepción 2)
              {Gestión de la excepción 2}
          catch (Excepción 3 y F)
              {Gestión de la excepción 3 y el resto}
}
```

Se puede dar el caso donde un imprevisto impida el cierre, generalmente por la interrupción del proceso, y, por tanto, la consiguiente liberación de recursos. Con el fin de que siempre se produzca la liberación de recursos es necesario que se añada el bloque **finally {}**, de modo que siempre se liberen los recursos, incluso cuando el proceso no finaliza correctamente.

```
public static void main(String[] args) {

    float resultado = 0;
    int dividendo = 10;
    int divisor = 0;

    try {
        System.out.println("- Inicio de una división");
        resultado = dividendo / divisor;
        System.out.println("El resultado es: "+ resultado);
    } catch (ArithmeticException e1) {
        System.out.println("Se ha producido un error aritmético");
        System.out.println("Se ha intentado hacer la siguiente división: "+ dividendo +"/"+ divisor);
    } finally {
        System.out.println("- Fin de la división");
    }
}
```

```
- Inicio de una división
Se ha producido un error aritmético
Se ha intentado hacer la siguiente división: 10/0
- Fin de la división
```

Imagen 5. Bloque try/catch/finally para captura de excepciones

Podemos incluir el método close () dentro del bloque finally {} si deseamos implementar una interfaz Closeable o AutoCloseable, con el fin de gestionar recursos.

```
for{
      try { cuerpo }
          catch (Excepción 1)
              {Gestión de la excepción 1}
          catch (Excepción 2)
              {Gestión de la excepción 2}
          catch (Excepción 3 y F)
              {Gestión de la excepción 3 y el resto}
}
Finally {
    > // XX.close()
}
```

2.6. Formas de acceso a los ficheros

La apertura de un fichero consta en crear el fichero si no existe y abrirlo para su lectura o escritura. Dependiendo de la acción la apertura también determina desde donde empieza a escribir o leer.

El cierre consta de terminar de usar el fichero, liberando la memoria, punteros, cerrando el flujo de datos y otras referencias.

La apertura y cierre de ficheros dependerá del modo de acceso.

Modo de acceso secuencial

Para este tipo de acceso se podrá usar las clases *FileReader*, *FileWriter*, *BufferedReader* y *BufferedWriter* para ficheros tipo texto y *FileInputStream*, *FileOutputStream*, *BufferedInputStream* y *BufferedOutputStream* para ficheros tipo binario.

- Para la **apertura**:Si no existe el fichero, se creará y se abrirá para la escritura.
- Si existe el fichero se abrirá para la escritura. Y si se indicó previamente el argumento *append* como true, la escritura empezará al final del fichero. De lo contrario el fichero será sobrescrito.

Para el **cierre** solo se invocará el método `close()`.

Se pueden ver los ejemplos de apertura y cierre en el apartado 12.1.4.

Modo de aleatorio

El acceso aleatorio solo se puede realizar para tipo de ficheros binarios, y se usará la clase *RandomAccessFile*.

Se podrá crear objetos de RandomAccessFile a partir de la ruta o un objeto File y añadiremos el modo, si es para lectura (r) o lectura y escritura (rw).

```
String ruta = "C:\\documentos\\configuracion\\config.bin";
RandomAccessFile raf = new RandomAccessFile(ruta, "rw");

raf.writeChars("HOLA MUNDO");

raf.seek(0);
for (int i = 0; i < raf.length() / 2; i++)
    System.out.print("" + raf.readChar());

raf.close();
```

Imagen 6. Ejemplo de instanciación de RandomAccessFile

2.7. Operaciones sobre ficheros con Java

Independientemente del tipo de fichero y acceso las operaciones básicas son idénticas.

El acceso a un fichero se realiza con un puntero y un *buffer* como medio de memoria. El puntero siempre debe señalar uno de los *bytes* del fichero o le sección denominada EOF, end of file, que determina la posición justo después del último *byte*.

Las operaciones básicas que podemos realizar con un fichero son:

- **Apertura.** La apertura del fichero se realiza creando una instancia con la que trabajaremos.
- **Lectura.** Usando el método `read()` podemos leer los contenidos de un fichero al volcarlos en una memoria. El puntero se sitúa justo detrás del último *byte* leído.
- **Salto.** Con el método `skip()` podemos avanzar hacia delante un número de bytes determinados.
- **Escritura.** Empleamos el método `write()` para escribir nuevo contenido en un fichero en el lugar deseado. El puntero se sitúa justo después del último *byte* leído.
- **Cierre.** Usando `close()` podemos cerrar el fichero.

La diferencia entre un acceso secuencial y uno aleatorio proviene de la posibilidad de mover el puntero. Con el primero el puntero solo se moverá al realizar alguna de las opciones anteriores, mientras que con el segundo podremos mover el puntero con libertad.

2.7.1. Operaciones de lectura

La lectura se debe llevar a cabo mediante un *buffer* el cuál será el que mostrará los caracteres para la lectura. El *buffer*, si no se especifica un máximo, mostrará tanto como pueda, Cuando el fichero muestre EOF no se mostrará nada.

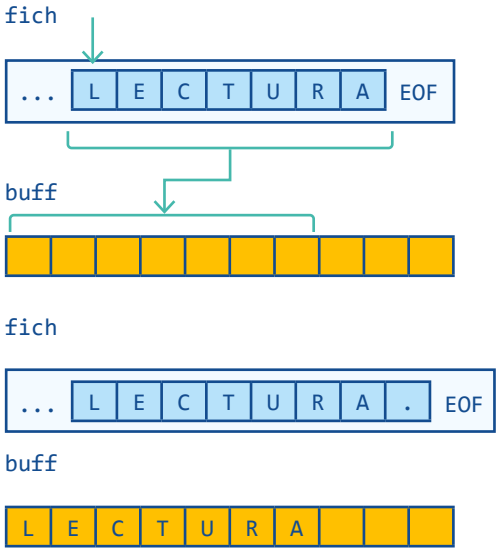

Imagen 7. Lectura de fichero

2.7.2. Operaciones de escritura

Cuando se escribe en un fichero se realiza también desde el *buffer*. Se debe especificar dónde se debe insertar los bytes recogidos en el *buffer*.

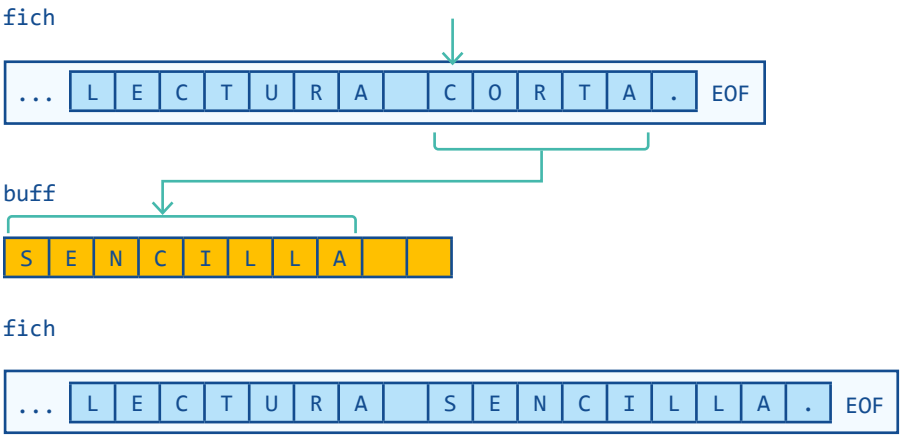

Imagen 8. Escritura dentro de un fichero

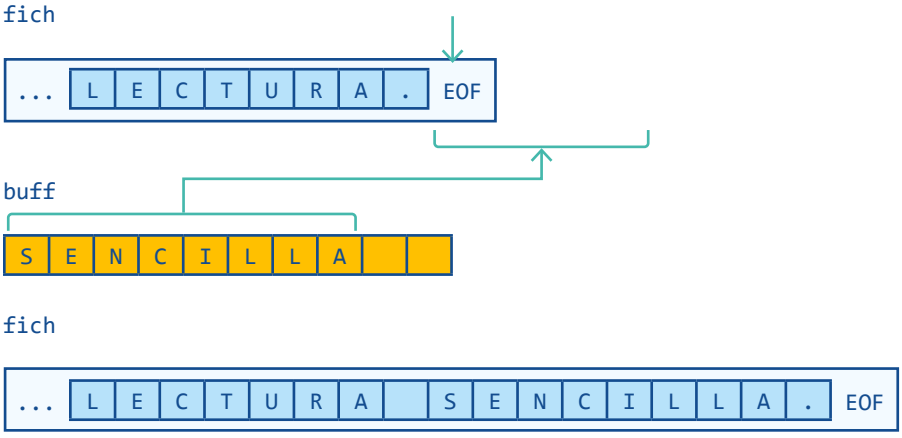

Imagen 9. Escritura al final de un fichero

2.8. Acceso secuencial a ficheros Java

Podemos emplear flujos para llevar a cabo operaciones en ficheros secuenciales usando paquetes `java.io`. Podemos considerar como una abstracción de alto nivel al flujo en relación con los ficheros, aunque los ficheros secuenciales también se consideran flujos, al igual que los datos transmitidos en la red o la entrada o salida de información dentro de un programa en su proceso de ejecución.

El paquete `java.io` permite, empleando la herencia, otorgar clases con la funcionalidad de entrada y salida y otras específicas en función del tipo de flujo empleado.

En relación con los cuatro tipos, que se conforman con la unión de flujos binarios y de texto con flujos de entrada y salida, podemos encontrar una jerarquía de clases.

2.8.1. Clases relacionadas con flujos de datos

En cualquier programa existe una interacción con elementos externos para la entrada de información y/o salida. Por ejemplo, cualquier entrada de información puede ser por teclado, fichero, red, etc. y de salida, por pantalla, fichero, etc.

Todos los lenguajes de programación ofrecen herramientas o mecanismos para poder hacer uso de elementos de entrada y salida. Estas herramientas trabajan sobre flujos de datos, que son una secuencia ordenada de datos que se transmiten desde una fuente hacia un destino, en Java se denomina ***stream.***

Para poder realizar una acción en un fichero es necesario que se cree un flujo, *stream*, asociado y acorde a la acción y al fichero empleado.

En java disponemos de funcionalidad para trabajar con flujos. En el paquete *java.io* se encuentran todas las clases que implementan la funcionalidad de entrada y salida para abstraernos y no tener que configurar los dispositivos.

Clases para el manejo de flujos de bytes

Todas las clases de manejo de flujos de byte heredan de las clases InputStream y OutputStream. A continuación, se describen las principales clases.

- **FileOutputStream y FileInputStream:** permite escribir y leer bytes en un fichero binario.

Escritura

```
char[] vocales = { 'A', 'E', 'I', 'O', 'U' };

String rutaFichero = "C:\\documentos\\fichero.bin";

FileOutputStream fos = new FileOutputStream(rutaFichero);
for (int i = 0; i < vocales.length; i++) {
    fos.write((byte) vocales[i]);
}

fos.close();
```

fichero.bin

```
1  AEIOU
2
```

Lectura

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.bin";

FileInputStream fis = new FileInputStream(rutaFichero);
int i;
while ((i = fis.read()) != -1) {
    System.out.print((char) i);
}

fis.close();
```

Console

```
<terminated> Fluj
AEIOU
```

Imagen 10. Ejemplo de FileOutputStream y FileInputStream

> **BufferedOutputStream y BufferedInputStream:** se usa un buffer intermedio para mejorar el rendimiento.

Escritura

```
char[] vocales = { 'A', 'E', 'I', 'O', 'U' };
String rutaFichero = "C:\\documentos\\fichero.bin";

FileOutputStream fos = new FileOutputStream(rutaFichero);
BufferedOutputStream bos = new BufferedOutputStream(fos);
for (int i = 0; i < vocales.length; i++) {
    bos.write((byte) vocales[i]);
}

bos.close();
fos.close();
```

fichero.bin

```
1  AEIOU
2
```

Lectura

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.bin";

FileInputStream fis = new FileInputStream(rutaFichero);
BufferedInputStream bis = new BufferedInputStream(fis);

int i;
while ((i = bis.read()) != -1) {
    System.out.println((char) i);
}

bis.close();
fis.close();
```

Console

```
<terminated> Flujo
A
E
I
O
U
```

Imagen 11. Ejemplo de BufferedOutputStream y BufferedInputStream

> **DataOutputStream y DataInputStream:** en un fichero binario todos sus datos están representados por bytes, con esta clase, es posible trabajar directamente con tipo de datos primitivos sobre un fichero binario.

Escritura

```
//Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero="C:\\documentos\\fichero.bin";

FileOutputStream fos = new FileOutputStream(rutaFichero);
DataOutputStream dos = new DataOutputStream(fos);
dos.writeChars("AEIOU");

dos.close();
fos.close();
```

fichero.bin

```
1  NUL A NUL E NUL I NUL O NUL U
```

Lectura

```
String rutaFichero="C:\\documentos\\fichero.bin";

FileInputStream fis = new FileInputStream(rutaFichero);
DataInputStream dis = new DataInputStream(fis);

while(dis.available()>0) {

    System.out.print(dis.readChar());
}

dis.close();
fis.close();
```

Console

```
<terminated> Flujo
AEIOU
```

Imagen 12. Ejemplo de DataOutputStream y DataInputStream

> **ByteArrayOutputStream y ByteArrayInputStream:** con estas clases podemos trabajar directamente con *arrays* de Byte y luego cargarlas al flujo para que se escriba o se lea. Para este tipo de clases hace falta invocar el método flush() para consolidar los datos escritos en el fichero.

```
char[] vocales = { 'A', 'E', 'I', 'O', 'U' };
String rutaFichero = "C:\\documentos\\fichero.bin";

FileOutputStream fos = new FileOutputStream(rutaFichero);
ByteArrayOutputStream bos = new ByteArrayOutputStream();

for (int i = 0; i < vocales.length; i++) {
    bos.write((byte) vocales[i]);
}

bos.writeTo(fos);
bos.flush();
bos.close();
fos.close();
```

fichero.bin

```
1  AEIOU
2
```

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.bin";

FileInputStream fis = new FileInputStream(rutaFichero);
ByteArrayInputStream bis = new ByteArrayInputStream(fis.readAllBytes());

int contenido = 0;
while ((contenido = bis.read()) != -1) {
    System.out.print((char) contenido);
}

bis.close();
fis.close();
```

Console

```
<terminated> Flujo
AEIOU
```

Imagen 13. Ejemplo de ByteArrayOutputStrem y ByteArrayInputStream

> **PrintStream:** permite agregar la capacidad de imprimir datos a un flujo de bytes de datos concreto.

```
String rutaFichero = "C:\\documentos\\fichero.bin";

FileOutputStream fos = new FileOutputStream(rutaFichero);
PrintStream ps = new PrintStream(fos);
ps.println("AEIOU");
ps.printf("AE%sOU", 'I');
ps.close();
fos.close();

ps = new PrintStream(System.out);
ps.println("AEIOU");

ps.close();
fos.close();
```

fichero.bin

```
1  AEIOU
2  AEIOU
3
```

Console

```
<terminated> Flujo
AEIOU
```

Imagen 14. Ejemplo de PrintStream

Clases para el manejo de flujos de caracteres

Todas las clases que manejan flujos de caracteres heredan de *Writer* y *Reader*. A continuación, se describen las principales clases.

- **FileWriter y FileReader:** permite escribir y leer de un fichero de texto.

Escritura

```
String rutaFichero = "C:\\documentos\\fichero.txt";

FileWriter fw = new FileWriter(rutaFichero);
fw.write("Hola Mundo");
fw.close();
```

fichero.txt

```
1  Hola Mundo
```

Lectura

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.txt";

FileReader fr = new FileReader(rutaFichero);

int i;
while ((i = fr.read()) != -1) {
    System.out.print((char) i);
}

fr.close();
```

Console

```
<terminated> FlujoC
Hola Mundo
```

Imagen 15. Ejemplo deFileWriter y FileReader

- **BufferedWriter y BufferedReader:** igual que la anterior, pero haciendo uso de un buffer intermedio para mejorar el rendimiento.

Escritura

```
String rutaFichero = "C:\\documentos\\fichero.txt";

FileWriter fw = new FileWriter(rutaFichero);
BufferedWriter bw = new BufferedWriter(fw);

bw.write("Hola Mundo");
bw.newLine();
bw.write("Uso de bufferedWriter");
bw.close();
fw.close();
```

fichero.txt

```
1  Hola Mundo
2  Uso de bufferedWriter
```

Lectura

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.txt";

FileReader fr = new FileReader(rutaFichero);
BufferedReader br = new BufferedReader(fr);

int i;
while ((i = br.read()) != -1) {
    System.out.print((char) i);
}

br.close();
fr.close();
```

Console

```
<terminated> FlujoCaracteres
Hola Mundo
Uso de bufferedWriter
```

Imagen 16. Ejemplo de BufferedWriter y BufferedReader

- **CharArrayWriter y CharArrayReader:** con estas clases podemos trabajar directamente con *arrays* de caracteres y luego cargarlas al flujo para que se escriba o se lea. Para este tipo de clases hace falta invocar el método *flush* para consolidar los datos escritos en el fichero.

Escritura

```
char[] array = {'H','o','l','a',' ','M','u','n','d','o'};

String rutaFichero = "C:\\documentos\\fichero.txt";

FileWriter fw = new FileWriter(rutaFichero);
CharArrayWriter caw = new CharArrayWriter();
caw.write(array);
caw.writeTo(fw);

caw.close();
fw.close();
```

fichero.txt

```
1  Hola Mundo
```

Lectura

```
String rutaFichero = "C:\\documentos\\fichero.txt";

FileReader fr = new FileReader(rutaFichero);
char[] lista = new char[20];
int i, x = 0;
while ((i = fr.read()) != -1) {
    lista[x] = ((char) i);
    x++;
}

CharArrayReader car = new CharArrayReader(lista);

int contenido = 0;
while ((contenido = car.read()) != -1) {
    System.out.print((char) contenido);
}

car.close();
fr.close();
```

Console

```
<terminated> FlujoC
Hola Mundo
```

Imagen 17. Ejemplo de CharArrayWriter y CharArrayReader

- **PrintWriter:** tiene la capacidad de imprimir datos a un flujo de caracteres de datos concreto. Como se puede ver en la imagen, puede imprimir en un fichero o consola.

```
String rutaFichero = "C:\\documentos\\fichero.txt";

FileWriter fw = new FileWriter(rutaFichero);
PrintWriter pw = new PrintWriter(fw);

pw.println("Hola mundo");
pw.printf("Soy %s", "Pepe");

pw.flush();
pw.close();
fw.close();

pw = new PrintWriter(System.out);
pw.println("Hola mundo");
pw.printf("Soy %s", "Pepe");
pw.flush();
pw.close();
```

```
fichero.txt
1  Hola mundo
2  Soy Pepe
```

```
Console
<terminated> Flu
Hola mundo
Soy Pepe
```

Imagen 18. Ejemplo de PrintWriter

2.8.2. Clases para recodificación

Es posible recodificar los datos entre los flujos binarios y de texto, de modo que podamos emplear el deseado, pero para ello es necesario especificar la codificación a su constructor.

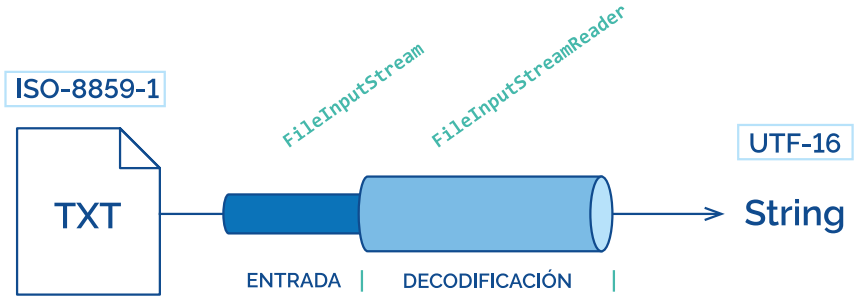

Imagen 19. Lectura desde un fichero que no emplea la codificación por defecto

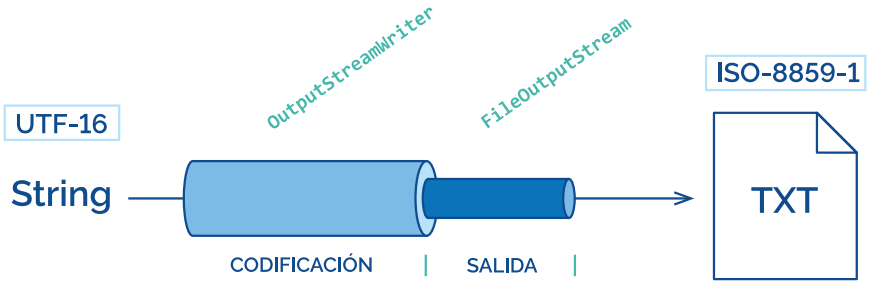

Imagen 20. Escritura en un fichero que no emplea la codificación por defecto

2.8.3. Clases para buffering

Ciertas clases emplean el *buffering*, el empleo de un *buffer*, para la aceleración de las operaciones de lectura y escritura.

El *buffer* representa la sección del fichero señalada por el puntero hasta que el archivo finaliza o el *buffer* se completa. Igualmente podemos escribir sobre él y trasladarlo al fichero, como ya hemos visto en puntos anteriores. La escritura mediante un *buffer* es llamada *flushing*.

	Clases con buffering para entrada y salida de flujo	
	Lectura	Escritura
Flujo binario	`BufferedInputStream`	`BufferedOutputStream`
Flujo de texto	`BufferedReader`	`BufferedWriter`

Métodos para lectura y escritura de líneas en clases para buffering para ficheros de texto		
Clase	Método	Funcionalidad
`BufferedReader`	String readLine()	Lee la línea actual
`BufferedReader`	Void newLine()	Escribe un separador de líneas

2.8.4. Operaciones de lectura para flujos de entrada

Las funciones `read` de las clases que son herederas de `InputStream` permiten la lectura de *bytes*, a diferencia de las que heredan `Reader` que permiten la lectura de caracteres. Pueden, mediante *buffer* leer hasta que este se llene o solo lo indicado. Con la función `skip` nos permite saltar un número determinados de *bytes*.

Los ficheros de texto pueden leer líneas empleando un `BufferedReader` a través de su función `readLine()`, siempre que se encuentre construido sobre un `FileReader`.

Lectura

```
// Tiene que existir previamente el fichero, si no, lanzará una excepción
String rutaFichero = "C:\\documentos\\fichero.txt";

FileReader fr = new FileReader(rutaFichero);
BufferedReader br = new BufferedReader(fr);

int i;
while ((i = br.read()) != -1) {
    System.out.print((char) i);
}

br.close();
fr.close();
```

```
Console ×  Problen
<terminated> FlujoCaracteres
Hola Mundo
Uso de bufferedWriter
```

Imagen 21. Ejemplo de BufferedReader

2.8.5. Operaciones de escritura para flujos de salida

Las funciones `write` de las clases que son herederas de `OutputStream` permiten la escritura de *bytes*, a diferencia de las que heredan `writer` que permiten la escritura de caracteres. Los primeros pueden, mediante *buffer* escribir en el fichero, o en el caso de llegar al final continuarán escribiendo y añadiendo *bytes*. Los segundos pueden, mediante `append`, escribir los contenidos de un `string`.

FileOutputStream y Writer puede poseer un parámetro que permite abrir un archivo y añadir contenido al final.

```
FileOutputStream(File file, boolean append)
FileOutputStream(String NOMBRE, boolean append)
Writer(File file, boolean append)
Writer(String NOMBRE, boolean append)
```

Métodos de escritura de clases para gestión de flujos de salida	
OutputStream	Writer
void write(int b)	void writer (int c)
void write(byte[] buffer)	void write(char[] buffer)
void write (byte[] buffer, int offset, int longitud)	void write (char[] buffer, int offset, int longitud)
	void write (String str)
	void write (String str, int offset, int longitud)
	Writer append(Char c)
	Writer append(CharSequence csq)
	Writer append(CharSequence csq, int offset, int longitud)
	BufferedWriter
	Void newLine()

Escritura

```
String rutaFichero = "C:\\documentos\\fichero.txt";

FileWriter fw = new FileWriter(rutaFichero);
BufferedWriter bw = new BufferedWriter(fw);

bw.write("Hola Mundo");
bw.newLine();
bw.write("Uso de bufferedWriter");
bw.close();
fw.close();
```

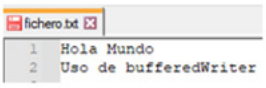

Imagen 22. Ejemplos de BufferedWriter

Usando flujo no es posible reemplazar contenido, para ello se debe abrir el propio documento o con un fichero auxiliar creado con `createTempFile()` de la clase `File`.

Estos tipos de ficheros se suelen crear en un directorio especial para ficheros.

2.9. Operaciones con ficheros de acceso aleatorio en Java

Para el acceso aleatorio a ficheros podemos emplear la clase `RandomAccessFile`, que junto a elementos ya mencionados añade:

- La función `seek()` permite situar el cursor en la posición deseada.
- Es posible realizar acciones de lectura y escritura sobre un fichero.

A pesar de estos añadidos aún quedan una multitud de acciones imposibles para los paquetes java.io como la eliminación o inserción de bloques de *bytes*.

Principales métodos de la clase RandomAccessFile

Método	Funcionalidad

`RandomAccessFile(File file, String mode)`
 	
`RandomAccessFile(String name, String mode)`
	Abre un fichero según sus permisos:
	+ **r:** modo de lectura.
	+ **rw:** modo de lectura y escritura.
	+ **rwd, rws:** Rw con escritura sincrónica.

`void close()`	Cierra el fichero.
`void seek(long pos)`	Posiciona el fichero donde se indique.
`int skipBytes(int n)`	Mueve el puntero el número de *bytes* especificado, o hasta el final si lo alcanza.

`int read()`
 	
`int read(byte[] buffer)`
 	
`int read(byte[] buffer, int offset, int longitud)`
	Lee los *bytes* indicados o hasta llenar el *buffer*.

`void readFully(byte[] buffer)`
 	
`readFully(byte[]buffer, int offset, int longitud)`
	`In read()`, pero con la excepción `IOExcepcion`.

`String readLine()`	Lee hasta el final de la línea actual.

`void write(int b)`
 	
`void write(byte[] buffer)`
 	
`void write(byte[] buffer, int offset, int longitud)`
	Escribe en el fichero el contenido del *buffer*, en la posición indicada o al final del fichero.

2.10. Organizaciones de ficheros

La organización de los ficheros permite estructurar los datos dentro de los ficheros con el fin de facilitar el desempeño correcto de estos. Es habitual la existencia de un campo, o clave, como modo de identificar los registros, lo cual permite diversos registros con los mismos valores para la clave.

Es posible la aparición de ficheros con estructuras complementarias como cabeceras o bloques de cierre, donde se puede añadir información sobre el propio fichero.

A pesar de todo esto los principales modelos de organización son:

- La organización secuencial.
- La organización secuencial indexada, ambas basadas en registros de longitud fija.

2.10.1. Organización secuencial

Los registros no poseen un orden dentro de los ficheros, y deben leerse uno a uno para encontrar el deseado.

A pesar de su sencillez este tipo de organización posee desventajas, como son:

- Búsquedas ineficientes, ya que se deben revisar uno a uno.
- El borrado de un registro implica tener que mover todos los registros posteriores una posición.

Como ventaja podemos decir que la inserción es muy sencilla, ya que solo debe añadirse al final.

2.10.2. Organización secuencial indexada

Resulta más laborioso en su creación, pues se deben indexar los índices, pero a cambio otorgara grandes comodidades para la búsqueda, permitiendo que podamos realizar búsquedas según diversos datos ordenados alfabéticamente.

La estructura de un índice está compuesta por dos columnas, una primera secuencial con el dato deseado y otra contigua que registra el fichero donde se encuentra dicho dato.

El número de índices posibles no está determinado, se pueden crear tantos índices como deseemos empleando los datos de los ficheros.

Con la inserción de un nuevo fichero se deberán reorganizar los índices para que coincidan con la realidad, por lo que un mayor número de índices implican una mayor labor en su reorganización. Otra desventaja es el aumento del espacio ocupado.

www.universae.com