Tema 8. Lectura y Escritura de Información

8.1. Entrada/salida de datos en Java

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar como para devolver los resultados obtenidos. La manera de representar estas entradas y salidas en Java es a base de **streams** (flujos de datos).

Un **stream** es un objeto que actúa de intermediario entre el programa y la fuente o destino de los datos. La información se traslada **en serie** (un carácter a continuación de otro) a través de este objeto. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Esta abstracción proporcionada por los flujos, hace que los programadores solo se tengan que preocupar de la forma de trabajar con estos objetos, sin importar el origen o destino concreto de los datos.

Por ejemplo, cuando se quiere mostrar algo en pantalla se hace a través de un **stream** que conecta el monitor al programa. Se da a ese **stream** la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet,…

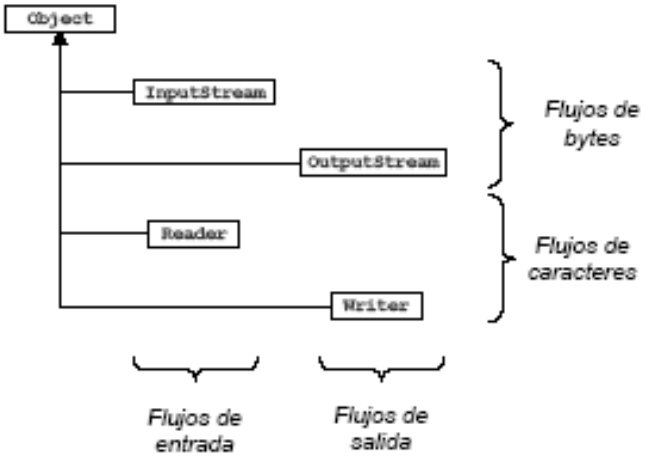
8.2. Clases de Java para lectura y escritura de datos

El paquete **java.io** contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este paquete existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal entre ellas radica en que una opera con **bytes** y la otra con **caracteres** (formados por dos bytes porque siguen el código **Unicode**). Además, dentro de cada familia hay una clase de entrada y otra de salida.

Las clases genéricas para manejar ambos tipos de datos son:

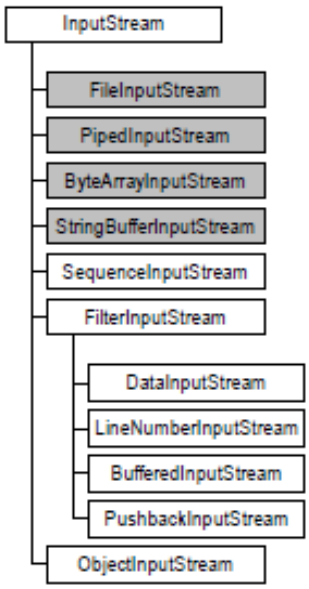
- Clases **Reader** y **Writer** para flujos de caracteres.
- Clases **InputStream** y **OutputStream** para flujos de bytes.

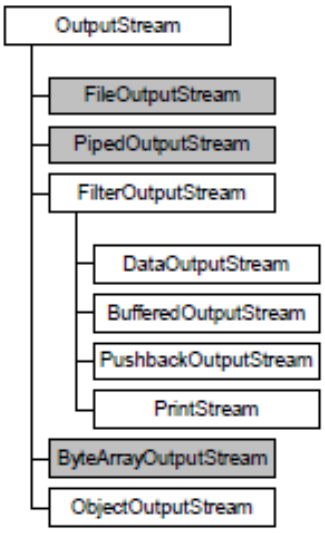
La siguiente figura muestra gráficamente estas 4 clases base del paquete **java.io**:



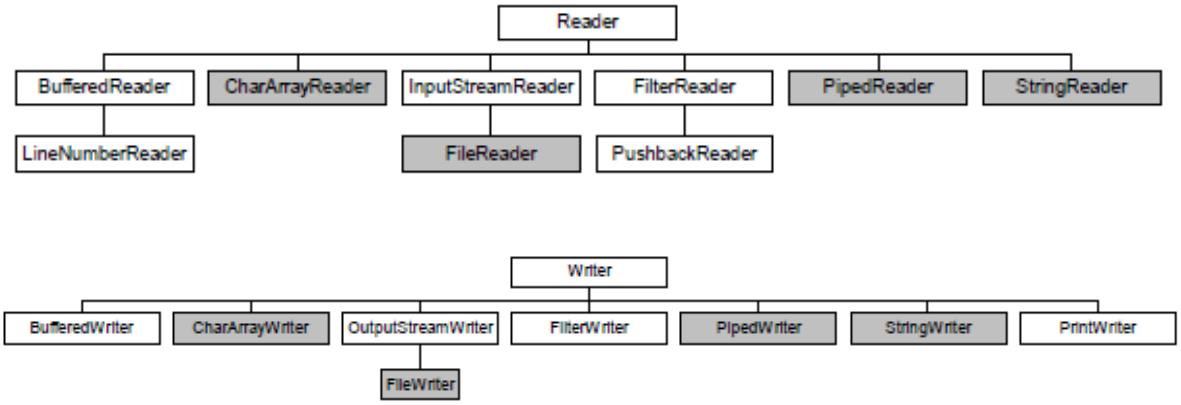
Desde **Java 1.0**, la entrada y salida de datos se podía hacer con clases derivadas de **InputStream** (para lectura) y **OutputStream** (para escritura). Estas clases tienen los métodos básicos **read()** y **write()** que manejan **bytes**, aunque no se suelen utilizar directamente.

Las siguientes figuras muestran las clases que derivan de **InputStream** y de **OutputStream**.





En **Java 1.1** aparecieron dos nuevas familias de clases derivadas de **Reader** y **Writer,** que manejan **caracteres** en vez de **bytes**. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las clases que heredan de **Reader y Writer** son las siguientes:



En las cuatro figuras anteriores las clases con **fondo gris** definen el dispositivo con que conecta el **stream**. Las demás clases (**fondo blanco**) añaden características particulares a la forma de enviar/recibir datos del stream. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader("archivo.txt"));
```

Con esta línea se ha creado un **stream** que permite leer del archivo **archivo.txt**. Además, se ha creado a partir de él un objeto **BufferedReader** que aporta la característica de utilizar **buffer**. Los caracteres que lleguen a través del **FileReader** pasarán a través del **BufferedReader**, es decir, utilizarán el **buffer**.

A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (**clases en gris**) y luego se le añadirán otras características (**clases en blanco**).

Se recomienda utilizar las clases **Reader** y **Writer** cuando se trabaje con texto. Algunas tareas como la **serialización** y la **compresión** necesitan las clases **InputStream** y **OutputStream.**

- **Los nombres de las clases de java.io**

Las clases del paquete **java.io** siguen una nomenclatura que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la siguiente tabla:

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

- **Clases que indican el origen o destino de los datos**

La siguiente tabla explica el uso de las clases que definen el lugar con que conecta el **stream**.

Clases	Función que realizan
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en **archivos** de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. También puede comunicar a dos programas distintos.

- **Clases que añaden características**

La siguiente tabla explica las funciones de las clases que alteran el comportamiento de un **stream** ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. *BufferedReader* por ejemplo tiene el método *readLine()* que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertenecen al mecanismo de la **serialización** y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc.

8.3. La clase IOException

Muchas de las operaciones realizadas por las clases del paquete **java.io** pueden lanzar una excepción **IOException**. Esta clase es la clase base de todas las excepciones que representan problemas cuando se realizan operaciones de entrada/salida.

IOException se trata de una excepción que **obligatoriamente hay que manejar**, por lo que debe ser capturada (con **try-catch**) o propagada (con **throws**).

Debido a que las operaciones de entrada/salida tratan con recursos externos pueden surgir muchos problemas. Por ejemplo, un fichero del cual queremos leer podría no existir o cuando intentamos abrir un fichero para actualizarlo se podría producir una excepción por no encontrarse el fichero. Todas estas situaciones deben tenerse muy en cuenta en el diseño de los programas para que sean lo más robustos posibles.

8.4. Clase File

Un objeto de la clase **File** puede representar un **archivo** o un **directorio**. Tiene los siguientes constructores:

File(String nombre**)**
File(String directorio, String nombre**)**
File(File directorio, String nombre**)**

Ejemplos:

```
File f1 = new File("c:\\windows\\notepad.exe"); // Un archivo
File f2 = new File("c:\\windows"); // Un directorio
File f3 = new File(f2, "notepad.exe"); // Es igual a f1
```

Como vemos para representar el carácter separador "\" hay que escaparlo previamente con otro carácter "\".

Además, podemos utilizar **rutas absolutas** y **rutas relativas**. La **ruta absoluta** se indica desde la unidad de disco en la que se está trabajando y la **relativa** desde el directorio actual.

Es importante tener en cuenta que cuando se crea un objeto **File** no se crea un fichero o directorio, únicamente se crea un objeto que representa al fichero o directorio, el cual puede o no existir. De hecho, al crear un objeto **File** no se realiza ningún tipo de comprobación de la existencia del fichero o directorio, ni de la legalidad de su nombre, por lo que para saber si existe se puede usar el método **boolean exists()*.***

- ### El problema de las rutas

Cuando se crean programas en Java hay que tener muy en cuenta que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la utilización de rutas sea problemática porque la forma de denominar rutas es distinta en cada sistema operativo.

Por ejemplo en **Windows** se puede utilizar como separador de carpetas la barra **"/"** o la doble barra invertida **"\\"**, mientras que en muchos sistemas **Unix** sólo es posible utilizar la barra **"/"**. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta se elige desde un cuadro de diálogo. También se pueden utilizar las **variables estáticas** de la clase **File**:

propiedad	uso
char **separatorChar**	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es "/" y en Windows es "\", que se debe escribir como \\, ya que el carácter \ permite colocar caracteres de control, de ahí que haya que usar la doble barra.
String **separator**	Como el anterior pero en forma de String
char **pathSeparatorChar**	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser ":", en Windows es ";"
String **pathSeparator**	Como el anterior, pero en forma de String

Ejemplo:

```
File f4 = new File("temp" + File.separator + "xyz.txt");
// Un archivo con ruta relativa
```

Una vez construido un objeto **File** se pueden utilizar sus métodos para obtener información o para modificar el fichero o directorio al que representa.

Si **File** representa un **archivo,** que existe, se pueden utilizar los siguientes métodos:

Métodos	Función que realizan
boolean isFile()	true si el archivo existe
long length()	tamaño del archivo en bytes
long lastModified()	fecha de la última modificación
boolean canRead()	true si se puede leer
boolean canWrite()	true si se puede escribir
delete()	borrar el archivo
RenameTo(File)	cambiar el nombre

Si **File** representa un **directorio** se pueden utilizar los siguientes métodos:

Métodos	Función que realizan
boolean isDirectory()	true si existe el directorio
mkdir()	crear el directorio
delete()	borrar el directorio
String[] list()	devuelve los archivos que se encuentran en el directorio

Además disponemos de los siguientes métodos que devuelven el **path** del objeto **File** de diferentes formas:

Métodos	Función que realizan
String getPath()	Devuelve el path que contiene el objeto File
String getName()	Devuelve el nombre del archivo
String getAbsolutePath()	Devuelve el path absoluto (juntando el relativo al actual)
String getParent()	Devuelve el directorio padre

Estos son algunos de los métodos más utilizados de la clase **File**, pero dispone de más, los cuales se pueden consultar en el **API de Java**.

Ejemplos:

- **Archivo1.java:** programa que muestra características de archivos/directorios.
- **Archivo2.java:** programa que muestra el nombre de los archivos/directorios de un directorio dado.
- **Archivo3.java:** programa que muestra el contenido de un directorio. Para cada elemento se muestra su tipo (directorio <DIR> o nada si es un archivo), fecha de última modificación y nombre.
- **Archivo4.java**: ejemplo de creación de objetos File.
- **Archivo5.java:** programa que crea un directorio, un archivo en él y después borra ambos.
- **Archivo6.java:** programa que crea un archivo temporal.

8.5. Trabajando con ficheros

Cuando se quiere conseguir persistencia de datos en el desarrollo de aplicaciones los ficheros entran dentro de las soluciones más sencillas. Vamos a ver algunas alternativas para trabajar con ficheros.

No son todas, solo son algunas. Sin embargo, estas soluciones sí pueden considerarse una referencia dentro de las soluciones actuales respecto al almacenamiento de datos en ficheros.

• Tipos de ficheros

Según el tipo de contenido:

- **Ficheros de caracteres** (o de texto): pueden ser creados y visualizados utilizando cualquier editor de texto que ofrezca el sistema operativo.

- **Ficheros binarios** (o de bytes): no contienen caracteres reconocibles, sino que los bytes que contienen representan otra información: imágenes, música, vídeo, etc. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que entiendan cómo están dispuestos los bytes dentro del fichero, y así poder reconocer la información que contiene.

Según la forma de acceso:

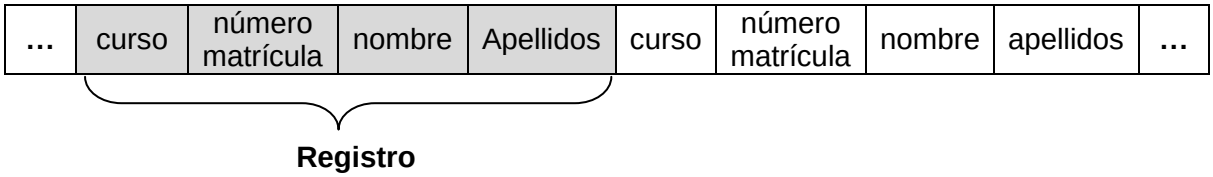
- **Ficheros secuenciales**: en este tipo de ficheros la información es almacenada como una secuencia de bytes o de caracteres, de forma que para acceder al byte o carácter i-ésimo es necesario pasar por todos los anteriores.

- **Ficheros directos** o **aleatorios**: a diferencia de los anteriores, se puede acceder directamente a una posición concreta del fichero, sin necesidad de pasar por todos los datos anteriores.

• Concepto de registro

Un registro es una agrupación de datos. Por ejemplo, en un programa que gestione alumnos, un registro podría ser la agrupación de [número de matrícula, nombre, apellidos y curso]. Cada uno de estos elementos se denomina **campo**. Estos campos pueden ser tipos elementales (int, char, …) o bien pueden ser otras estructuras de datos, incluso referencias a objetos.

A continuación podemos ver gráficamente como es la estructura de un registro:



Los registros, como se puede observar en la figura, son una agrupación generalmente heterogénea de campos, tanto por sus tipos de datos como por su contenido. No obstante, pueden existir registros que contengan solamente un campo y, en ese caso, la estructura sería homogénea.

Si nos centramos en la programación orientada a objetos, lo normal es que en un fichero almacenemos objetos y no registros, y que en vez de almacenar campos almacenemos atributos (**serialización**)

- ## Operativa

La forma de trabajar con ficheros en un programa suele ser siempre la misma, ya sea en Java u otros lenguajes de programación:

1º. Se abre el fichero
2º. Se accede al interior del fichero (para leer o escribir)
3º. Si ya no se va a necesitar más el fichero se cierra

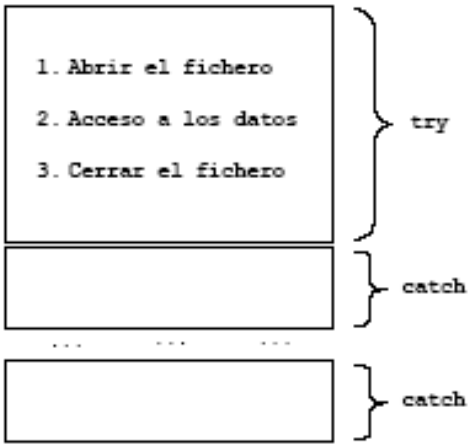
Las operaciones básicas que se realizan en un fichero son:

- Añadir nuevos datos
- Consultar información
- Modificar datos
- Eliminar datos

Son las llamadas operaciones **CRUD *(Create, Read, Update, Delete)***, que son las funciones básicas en bases de datos o capa de persistencia de una aplicación.

La operación de **apertura de un fichero** es costosa desde el punto de vista del rendimiento, por lo que el fichero deberá permanecer abierto mientras se prevea que se va a acceder a él.

Por otro lado, en Java para **tratar las excepciones** generadas al trabajar con ficheros suele crearse un gran bloque **try** al inicio del código en el que se accede al fichero. En este bloque de código se abre, accede y cierra el fichero ya que este tipo de sentencias generan todas los mismos tipos de excepciones. Las sentencias **catch** se añaden tras el bloque **try** desde las más específicas a las más genéricas. La siguiente figura muestra esta estructura:



Debe también tenerse muy en cuenta que las variables declaradas dentro del bloque **try** no son accesibles fuera de él, en particular ni en los bloques **catch** ni en el resto del código, por lo que si se va a utilizar una variable fuera del bloque **try** hay que declararla por encima de él.

8.6. Ficheros de acceso secuencial

Un fichero secuencial es una sucesión de registros **almacenados consecutivamente**, de tal forma que para acceder a un registro **n** es obligatorio pasar por los **n-1** registros anteriores. En este tipo de ficheros los registros se almacenan en el **orden de llegada**.

8.6.1. Lectura / Escritura en un fichero secuencial

- **Lectura y escritura de bytes**

Se utilizan dos clases que descienden de **InputStream** y **OutputStream**:

- **FileInputStream** (para lectura)
- **FileOutputStream** (para escritura)

Estas clases disponen de métodos **read** y **write** para leer y escribir, pero es más interesante utilizar las clases **DataInputStream** y **DataOutputStream** que están mucho más preparadas para leer y escribir **datos primitivos de todo tipo**.

Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de **bytes**. Por ello se utilizan para almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).

A continuación se describen los pasos a seguir para escribir y leer bytes de un fichero:

Proceso de Escritura:

1. Crear un objeto **FileOutputStream**[1] indicando la ruta al fichero o un objeto **File** que representa al fichero donde se quiere escribir.
2. Crear un objeto **BufferedOutputStream** a partir del objeto anterior. Esto es conveniente ya que se añade un **buffer** al manejo de los datos.
3. Crear un objeto **DataOutputStream** a partir del objeto anterior.
4. Usar el objeto **DataOutputStream** para escribir los datos mediante los métodos **write*tipo*** donde *tipo* es el tipo de datos a escribir (int, double, ...). A este método se le pasa como único argumento los datos a escribir.
5. Cerrar el archivo mediante el método **close** del objeto **DataOutputStream**.

Ejemplo: **EscribirBytes.java**

```
File f = new File(".\\Ficheros\\prueba.dat");
Random r = new Random(); // objeto para crear aleatorios

try {

      // abrimos el fichero, si no existe se crea
      DataOutputStream dos = new DataOutputStream(
                  new BufferedOutputStream(
                  new FileOutputStream(f)));

      // generamos 200 double aleatorios y les escribimos
      // en el fichero
      for (int i = 0; i < 199; i++) {
            dos.writeDouble(r.nextDouble()); // escritura
      }

      dos.close(); // cerramos el fichero

} catch (IOException e) {
      System.out.println("Error de escritura en el archivo");
}
```

[1] Al crear un objeto de este tipo se abre el fichero para escritura, de forma que si no existe lo crea, y si ya existe se sobrescribe su contenido. En el constructor de **FileOutputStream** se puede indicar un segundo parámetro booleano, que con valor **true** abre el fichero para añadir más datos no sobrescribiendo su contenido (si ya existía el fichero).

Proceso de Lectura:

1. Crear un objeto **FileInputStream**[2] indicando la ruta al fichero o un objeto **File**.
2. Crear un objeto **BufferedInputStream** a partir del objeto anterior.
3. Crear un objeto **DataInputtream** a partir del objeto anterior.
4. Usar el objeto **DataInputStream** para leer los datos mediante los métodos **read*tipo*.** Estos métodos devuelven el dato leído.
5. Cerrar el archivo mediante el método **close** del objeto **DataInputStream**.

Como vemos, el proceso es análogo a la escritura, sólo que en este caso hay que tener en cuenta que al leer se puede alcanzar el final del fichero produciéndose entonces una excepción del tipo **EOFException** (que es subclase de **IOException**), por lo que habrá que controlarla.

Ejemplo: **LeerBytes.java**

```
boolean finArchivo = false; //controla el final del fichero
double d; // almacena el dato leído

File f = new File(".\\Ficheros\\prueba.dat");

try {
      DataInputStream dis = new DataInputStream(
                  new BufferedInputStream(
                  new FileInputStream(f)));

   while (!finArchivo) { //mientras haya datos
         d = dis.readDouble();  //leemos dato (double)
         System.out.println(d); //lo mostramos por pantalla
   }

            dis.close();  //cerramos el fichero

} catch (EOFException e) {
      finArchivo = true;
} catch (FileNotFoundException e) {
      System.out.println("Error. No se puede abrir un fichero inexistente");
} catch (IOException e) {
      System.out.println("Error de lectura en fichero");
}
```

Bucle que se ejecuta de forma indefinida hasta que se lance la excepción EOFException

[2] Al crear un objeto **FileInputStream** se intenta abrir el fichero para lectura generando una excepción del tipo **FileNotFoundException** si el archivo no existe.

- **Lectura y escritura de caracteres**

Se utilizan las clases:
- **FileReader** (para lectura)
- **FileWriter** (para escritura)

Estas clases disponen de métodos **read** y **write** para leer y escribir, pero como ya se ha comentado en repetidas ocasiones es conveniente utilizar un buffer intermedio para mejorar el rendimiento, por lo que se utilizarán las clases **BufferedReader** y **BufferedWriter** para leer y escribir texto respectivamente.

A continuación se describen los pasos a seguir para escribir y leer texto de un fichero:

Proceso de Escritura:

1. Crear un objeto **FileWriter**[3] indicando la ruta al fichero o un objeto **File** que representa al fichero donde se quiere escribir.
2. Crear un objeto **BufferedWriter** a partir del objeto anterior.
3. Usar el objeto anterior para escribir los datos mediante el método **write()** al que se le pasa como único argumento el dato a escribir. Otros métodos característicos de **BufferedWriter** son **flush()** y **newline()**.
4. Cerrar el archivo mediante el método **close** del objeto **BufferedWriter**.

Ejemplo: **EscribirTexto.java**

```
BufferedReader teclado = new BufferedReader(new
            InputStreamReader(System.in));
String s; // almacena la frase introducida por teclado

try {
   FileWriter fw = new FileWriter(".\\Ficheros\\frases.txt",true);
   BufferedWriter bw = new BufferedWriter(fw);

   //pedimos frases por teclado y las escribimos en el fichero
   System.out.print("Introduce una frase: ");
   s = teclado.readLine();
   while (s.length() > 0){
      bw.write(s); //la almacenamos en el fichero
      bw.newLine();
      System.out.print("Introduce una frase: "); //nueva frase
      s = teclado.readLine();
```

[3] Al crear un objeto de este tipo se abre el fichero para escritura, de forma que si no existe lo crea, y si ya existe se sobrescribe su contenido. En el constructor de **FileWriter** se puede indicar un segundo parámetro booleano, que con valor **true** abre el fichero para añadir más datos no sobrescribiendo su contenido (si ya existía el fichero).

```
        }
        bw.close(); //cerramos el ficheros
    } catch (IOException e) {
        System.out.println("Error de entrada/salida");
    }
```

Proceso de Lectura:

1. Crear un objeto **FileReader**[4] indicando la ruta al fichero o un objeto **File**.
2. Crear un objeto **BufferedReader** a partir del objeto anterior.
3. Usar el objeto anterior para leer los datos mediante el método **readline()** que lee líneas de texto como un String. Éstas deben ser procesadas si se quiere extraer valores individuales.
4. Cerrar el archivo mediante el método **close** del objeto **BufferedReader**.

Ejemplo: **LeerTexto.java**

```
File f = new File(".\\Ficheros\\frases.txt"); //objeto File

try {
      FileReader fr = new FileReader(f);
      BufferedReader br = new BufferedReader(fr);
      String s; //almacena cada línea leída del fichero
      s = br.readLine(); //lectura anticipada
      while (s != null) { //mientras haya cadenas por leer
            System.out.println(s); //la mostramos por pantalla
            s = br.readLine(); // leemos otra línea del fichero
      }

      br.close(); //cerramos el fichero
} catch (FileNotFoundException e) {
      System.out.println("Error al abrir el archivo");
} catch (IOException e) {
      System.out.println("Error de entrada/salida");
}
```

Ejemplo:

- **Archivo8.java:** programa que copia el contenido de un fichero de texto en otro.

[4] Al crear un objeto **FileReader** se intenta abrir el fichero para lectura, generando una excepción del tipo **FileNotFoundException** si el archivo no existe.

8.6.2. Actualización de un fichero secuencial

La actualización de un fichero secuencial implica **añadir** nuevos registros, **modificar** datos de registros existentes o **borrarlos**; es lo que se conoce como altas, bajas y modificaciones.

- **Altas**

Para dar de alta un nuevo registro hay que abrir el fichero en modo **"append"** creando los objetos **FileOutputStream** o **FileWriter** (según corresponda) pasando **true** como segundo parámetro al constructor. De esta forma, se abre el fichero para añadir (no sobrescribir), y el nuevo registro se añade al final del fichero.

- **Bajas**

Para dar de baja un registro de un archivo secuencial se utiliza un **fichero auxiliar**, también secuencial. El proceso es el siguiente:

1. Mientras haya datos, leer registro del fichero original.

   1.1. Si el registro actual no se quiere dar de baja se escribe en el fichero auxiliar. Volver al punto 1.
   1.2. Si el registro actual se quiere dar de baja volver al punto 1.

   Una vez finalizada la lectura del fichero original se tienen dos ficheros: el original y el auxiliar (sin el registro dado de baja).

2. Eliminar el fichero original con método **delete()**.

3. Renombrar el fichero auxiliar con el nombre del fichero original, método **renameTo().**

- **Modificaciones**

La modificación de registros se realiza siguiendo los mismos pasos que en las bajas. Cada registro del fichero original se escribe tal cual en el fichero auxiliar hasta llegar al registro a modificar, en este momento se realizan los cambios oportunos y se graba en el fichero auxiliar el registro modificado. Se continúa el proceso hasta leer el último registro del fichero original. Finalmente, se borra el fichero original y se renombra el fichero auxiliar.

8.7. Ficheros de acceso aleatorio

Un fichero de **acceso aleatorio o directo** permite acceder directamente a un registro mediante la especificación de un **índice** que se corresponde con la posición del registro **respecto al origen del archivo**.

Un fichero de este tipo puede verse como un array de bytes con un **puntero** (**file pointer**) como si fuera el índice del array que puede ser movido como queramos.

La principal característica de los ficheros de acceso aleatorio radica en la rapidez de acceso a un determinado registro frente a la organización secuencial, que supone leer todos los registros anteriores. Conociendo el **índice** del registro se puede situar el **puntero** en la posición donde comienza el registro y a partir de esa posición realizar la operación de lectura o escritura. Cada vez que se lee **un byte** el puntero avanza automáticamente **una posición**.

Las operaciones que se realizan con los ficheros de acceso directo son las usuales: **creación**, **altas**, **bajas**, **consultas** y **modificaciones**.

Es imprescindible a la hora de trabajar con ficheros aleatorios **conocer el tamaño de los registros**, de forma que nos podamos mover de un registro a otro para después leer o modificar los datos. Todos los registros no tienen por qué ser del mismo tamaño, simplemente hay que poder determinar el **tamaño máximo** que pueden tener.

Para determinar el **tamaño máximo de un registro** se suma la máxima longitud de cada dato. Así, para los campos de tipo **String** se debe prever el máximo número de caracteres, además si se escribe en formato **UTF** se añaden 2 bytes más (para guardar la longitud de la cadena). Los campos de tipo primitivo tienen una longitud fija:

char: 2 bytes
int: 4 bytes
double: 8 bytes
...

8.7.1. Clase RandomAccessFile

La clase **RandomAccessFile** dispone de métodos para procesar archivos de acceso directo. Cuenta con los métodos de las clases **DataInputStream** y **DataOutputStream** para leer y escribir, además de métodos específicos.

Para crear un objeto **RandomAccessFile** podemos usar cualquiera de los siguientes constructores:

new RandomAccessFile(File fichero, String **modo**)
new RandomAccessFile(String fichero, String **modo**)

El 2º argumento es el modo de apertura del fichero. Puede tener los siguientes valores:

"r" **Modo de sólo lectura**. Sólo se pueden leer registros, no se pueden modificar o grabar registros.

"rw" **Modo de lectura/escritura**. Permite hacer operaciones tanto de lectura como de escritura de registros

Si el fichero no existe y se abre en modo de sólo lectura ("r") se obtendrá una **FileNotFoundException**, y si se abre en modo de lectura y escritura ("rw") se crea un fichero de longitud cero.

Ejemplo: queremos consultar el fichero Libros.dat con acceso directo:

```
File f=new File("Libros.dat");
try {
   RandomAccessFile raf=new RandomAccessFile(f,"r");
   ...
}catch (FileNotFoundException e) {
   System.out.println("Error al abrir el fichero");
}catch (IOException e) {
   System.out.println("Error de entrada/salida");
}
```

- ## Métodos de posicionamiento

Cuando se habla del **puntero del fichero** se hace referencia a la posición (en **nº de bytes**) a partir de la cual se va a realizar la siguiente operación de lectura o escritura. Una vez realizada la operación, el puntero queda situado justo después del último byte leído o escrito. Si por ejemplo, el puntero se encuentra en la posición **n** y se lee un campo **int** y un campo **double**, la posición del puntero después de la lectura será **n+12**.

Algunos métodos fundamentales son: (todos pueden lanzar excepciones **IOException**)

✓ **long getFilePointer().** Devuelve la posición actual del puntero (en bytes). Ese número de bytes representa el número de bytes desde el inicio del fichero hasta la posición actual. Siguientes lecturas o escrituras se harán a partir de esta posición.

Por ejemplo, la siguiente llamada al método **getFilePointer** devuelve 0 porque inmediatamente después de abrir el fichero el puntero está situado al inicio del fichero, es decir, en el byte 0:

```
RandomAccessFile raf=new RandomAccessFile("Libros.dat","rw");
System.out.println("Posición del puntero: "
        + raf.getFilePointer());
```

- ✓ **void seek(long n).** Desplaza el puntero del fichero n bytes, tomando como origen el inicio del fichero (byte 0). Es el método más utilizado, ya que permite colocarse en una posición concreta del archivo.

  OJO!! Es erróneo pasar desplazamiento negativos al método seek, ya que éste es relativo al byte 0 del archivo. Si el desplazamiento que pasamos es mayor que el tamaño del fichero y a continuación realizamos una operación de escritura el fichero se amplía automáticamente.

- ✓ **long length**(). Devuelve el tamaño del fichero en bytes. Si queremos situar el puntero al final del fichero Libros.dat pondremos:

```
raf.seek(raf.length());
```

• Lectura y escritura

Los métodos para leer y escribir en un fichero directo son los mismos que los de las clases **DataInputStream** y **DataOutputStream**:

- ✓ **readBoolean, readByte, readChar, readInt, readDouble, readFloat, readUTF, readLine.** Métodos de lectura. Leen un dato del tipo indicado. En el caso de *readUTF* lee una cadena en formato Unicode.

- ✓ **writeBoolean, writeByte, writeBytes, writeChar, writeChars writeInt, writeDouble, writeFloat, writeUTF, writeLine.** Métodos de escritura**.** Todos reciben como parámetro el dato a escribir. Escriben encima de lo existente. Para escribir al final del fichero hay que colocar el puntero al final del fichero.

Como vemos **RandomAccessFile** soporta la lectura y escritura de todos los tipos de datos primitivos. Cada operación de lectura o escritura avanza el puntero del fichero tantas posiciones como el número de bytes leídos o escritos. Además, **RandomAccessFile** tiene su espacio de almacenamiento intermedio, así que no hay que añadirle un buffer adicional.

Cuando ya no se necesite trabajar con el fichero, éste debe ser cerrado con el método **close()** y así liberar la memoria y los recursos asociados.

Ejemplos:

- **Directo1.java:** programa que muestra como leer el contenido de un fichero directo de varias formas.

- **Personas.java + Registro.java**: programa que crea el fichero directo *Datos.dat* con información relativa a personas. El formato de un registro es el siguiente:

Nombre	cadena (máximo 20 caracteres)
Apellidos	cadena (máximo 40 caracteres)
Edad	entero
Estado Civil	carácter (S: soltero, C: casado, V: viudo)

8.8. Almacenamiento de objetos en ficheros. Persistencia. Serialización

La **serialización** de objetos consiste en transformar un objeto en una secuencia de bytes de tal manera que se represente el **estado** de dicho objeto. Una vez que tenemos serializado el objeto, esa secuencia de bytes pueden ser posteriormente leídos para restaurar el objeto original.

Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en un ordenador que corra bajo Windows, serializarlo y enviarlo a través de la red a una estación de trabajo que corra bajo UNIX donde será correctamente reconstruido. No tenemos que preocuparnos, en absoluto, de las diferentes representaciones de datos en los distintos ordenadores.

El **estado** de un objeto es, como ya sabemos, el valor de cada uno de sus campos. Imaginemos que un campo es a su vez otro objeto, en ese caso debería ser serializado para serializar el primer objeto.

La **serialización** es una característica del lenguaje Java para dar soporte a:

- ✓ La **invocación remota de objetos** (RMI)
- ✓ La **persistencia**

La **invocación remota de objetos** permite a los objetos que viven en otros ordenadores comportarse como si vivieran en nuestra propia máquina. La **serialización** es necesaria para transportar los argumentos y los valores de retorno.

La **persistencia**, es una característica importante de los **JavaBeans**. El estado de un componente es configurado durante el diseño. La **serialización** nos permite guardar el estado de un componente en disco, abandonar el IDE (Entorno Integrado de Desarrollo) y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

• La interfaz *Serializable*

Para poder serializar un objeto, es necesario que su clase implemente la interfaz **java.io.Serializable**. Esta interfaz no declara ningún método miembro, se trata de una

interfaz vacía, luego el propósito de esto es marcar las clases que vamos a convertir en secuencias de bytes.

Veamos un ejemplo muy sencillo:

```
public class Amigo implements java.io.Serializable {
    private String nombre;
    private String telefono;

    public Amigo(String nom, String tf){
        this.nombre=nom;
        this.telefono=tf;
    }

    public String imprimir(){
        return this.nombre + " -> " + this.telefono;
    }
}
```

La clase Amigo se ha marcado como serializable, ahora Java se encargará de realizar la serialización de forma automática

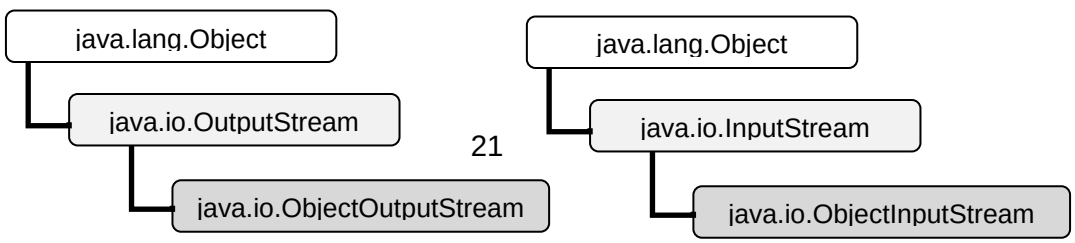
No tenemos que escribir ningún otro método. El método **defaultWriteObject** de la clase **ObjectOutputStream** realiza la **serialización** de los objetos de una clase. Este método escribe en el flujo de salida todo lo necesario para reconstruir dichos objetos:

- ✓ La clase del objeto
- ✓ La firma de la clase (class signature)
- ✓ Los valores de los miembros que no tengan los modificadores **static** o **transient**, incluyendo los miembros que se refieren a otros objetos.

El método **defaultReadObject** de la clase **ObjectInputStream** realiza la **deserialización** de los objetos de una clase. Este método lee el flujo de entrada y reconstruye los objetos de dicha clase.

• Almacenamiento y recuperación de objetos

Para el almacenamiento y recuperación de objetos utilizaremos las clases **ObjectOutputStream** y **ObjectInputStream**, cuyo manejo es análogo a las clases anteriormente vistas, **DataOutputStream** y **DataInputStream**.



Para escribir los objetos en el flujo de salida se utiliza el método **writeObject** del objeto **ObjectOutputStream**, al que se le pasa como parámetro el objeto a escribir.

De forma análoga, para leer los objetos del flujo de entrada, se utiliza el método **readObject** del objeto **ObjectInputStream**. En este caso será necesario realizar un **casting** para convertir los datos leídos al objeto de tipo deseado.

Para ver su funcionamiento, siguiendo con el ejemplo anterior, vamos a ver cómo se graban objetos `Amigo` en un fichero y cómo se recuperan posteriormente.

```
public class PruebaFichero {

    public static void main(String[] args) throws IOException {

        //escritura en el fichero de un objeto Amigo
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("amigos.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(new Amigo("Juan", "600111222"));
        } catch (FileNotFoundException ex) {
            System.out.println("Fichero no encontrado.");
        } finally {
            try {
                fos.close();
            } catch (IOException ex) {
                System.out.println("Se ha producido un error de entrada/salida.");
                ex.printStackTrace();
            }
        }



        //lectura del fichero de un objeto Amigo
        FileInputStream fis = null;
        try {
            File f = new File("amigos.txt");
            if (f.exists()) {
```

```
                fis = new FileInputStream(f);
                ObjectInputStream ois = new ObjectInputStream(fis);
                while (true) {
                    Amigo a = (Amigo) ois.readObject();
                    System.out.println(a.imprimir());
                }
            }
        } catch (EOFException ex) {
            System.out.println("Fin fichero.");
        } catch (ClassNotFoundException ex) {
            System.out.println("Fichero no encontrado.");
        } finally {
            try {
                fis.close();
            } catch (IOException ex) {
               System.out.println("Se ha producido un error de entrada/salida.");
                ex.printStackTrace();
            }
        }
    }
}
```

- **El modificador *transient***

Cuando un dato de una clase contiene información sensible, hay disponibles varias técnicas para protegerla. Incluso cuando dicha información es privada (modificador **private**) una vez que se ha enviado al flujo de salida alguien puede leerla en el archivo en disco o interceptarla en la red.

El modo más simple de proteger la información sensible es ponerla el modificador **transient**.

Veamos un ejemplo con la clase `Cliente` que se muestra a continuación:

```
public class Cliente implements java.io.Serializable {

    private String nombre;
    private transient String passWord;

    public Cliente(String nombre, String pw) {
        this.nombre = nombre;
        passWord = pw;
    }


    public String toString() {
```

```
        String texto = (passWord == null) ? "(no disponible)" : passWord;
        texto += nombre;
        return texto;
    }
}
```

La clase tiene dos miembros dato, el nombre del cliente y la contraseña. Además redefine el método **toString()**, que devolverá el nombre del cliente y la contraseña. En el caso de que el miembro **passWord** valga **null** se imprimirá el texto *"(no disponible)"*.

A continuación vamos a guardar un objeto de la clase `Cliente` en el archivo *clientes.obj.* Posteriormente, leeremos el archivo para reconstruir el objeto de dicha clase.

```
try {

    //escritura de un objeto Cliente en el fichero
    Cliente cliente = new Cliente("Angel", "xyz");

    ObjectOutputStream salida = new ObjectOutputStream(
        new FileOutputStream("clientes.obj"));
    salida.writeObject("Datos del cliente\n");
    salida.writeObject(cliente);
    salida.close();

    //lectura del objeto Cliente del fichero
    ObjectInputStream entrada = new ObjectInputStream(
        new FileInputStream("clientes.obj"));
    String str = (String) entrada.readObject();
    Cliente obj = (Cliente) entrada.readObject();
    System.out.println("------------------------------");
    System.out.println(str + obj);
    System.out.println("------------------------------");
    entrada.close();

} catch (ClassNotFoundException ex) {
    System.out.println("Fichero no encontrado.");
} catch (IOException ex) {
    System.out.println("Se ha producido un error de entrada/salida.");
    ex.printStackTrace();
}
```

La salida del programa es:

```
------------------------------
Datos del cliente
(no disponible)Angel
------------------------------
```

Como hemos comprobado, la información sensible guardada en el atributo *password* que tiene por modificador **transient** no ha sido guardada en el archivo. En la reconstrucción del objeto con la información guardada en el archivo, *password* toma el valor **null**.

- **Objetos compuestos**

Vamos a estudiar el caso de que un objeto tenga uno de sus miembros también objeto, y en este caso, como funciona la **serialización**.

Para ello definimos las clases `Rectangulo` y `Punto`.

```
public class Punto implements java.io.Serializable {

    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Punto() {
        x = 0;
        y = 0;
    }

    public void desplazar(int dx, int dy) {
        x += dx;
        y += dy;
    }

    public String toString() {
        return new String("(" + x + ", " + y + ")");
    }
}
```

La clase `Rectangulo` contiene un subobjeto de la clase `Punto`.

```
public class Rectangulo implements java.io.Serializable {

    private int ancho;
    private int alto;
    private Punto origen;

    public Rectangulo() {
        origen = new Punto(0, 0);
```

```
        ancho = 0;
        alto = 0;
    }

    public Rectangulo(Punto p) {
        this(p, 0, 0);
    }

    public Rectangulo(int w, int h) {
        this(new Punto(0, 0), w, h);
    }

    public Rectangulo(Punto p, int w, int h) {
        origen = p;
        ancho = w;
        alto = h;
    }

    public void desplazar(int dx, int dy) {
        origen.desplazar(dx, dy);
    }

    public int calcularArea() {
        return ancho * alto;
    }

    public String toString() {
        String texto = origen + " w:" + ancho + " h:" + alto;
        return texto;
    }
}
```

Como podemos apreciar, ambas clases implementan la interfaz **Serializable**.

A dichas clases se les ha añadido la redefinición del método **toString()** (esta redefinición no es necesaria aunque es ilustrativa para explicar el comportamiento de un objeto compuesto). Como podemos observar, en la definición de **toString()** de la clase `Rectangulo` se hace una llamada implícita a la función **toString()** de la clase `Punto`. La composición como vemos permite reutilizar el código existente.

Ahora vamos a guardar en un archivo un objeto de la clase `Rectangulo` y posteriormente lo reconstruimos:

```
Rectangulo r = new Rectangulo(new Punto(10, 10), 30, 60);

try {
```

```
        ObjectOutputStream salida = new ObjectOutputStream(new
              FileOutputStream("figuras.obj"));
        salida.writeObject("Guardar un objeto compuesto\n");
        salida.writeObject(r);
        salida.close();



        ObjectInputStream entrada = new ObjectInputStream(new
              FileInputStream("figuras.obj"));
        String str = (String) entrada.readObject();
        Rectangulo obj = (Rectangulo) entrada.readObject();
        System.out.println("------------------------------");
        System.out.println(str + obj);
        System.out.println("------------------------------");
        entrada.close();

} catch (ClassNotFoundException ex) {
        System.out.println("Fichero no encontrado.");
} catch (IOException ex) {
        System.out.println("Se ha producido un error de entrada/salida.");
        ex.printStackTrace();
}
```

En el caso de que nos olvidemos de implementar la interfaz **Serializable** en la clase `Punto` que describe el subobjeto de la clase `Rectangulo`, se lanza una excepción **java.io.NotSerializableException**.

- ## Serialización personalizada

El proceso de serialización proporcionado por el lenguaje Java es suficiente para la mayor parte de las clases, ahora bien, se puede personalizar para aquellos casos específicos.

Para personalizar la serialización, es necesario "sobrecargar" los métodos **writeObject** y **readObject** de aquella clase cuya serialización se quiere controlar. En realidad no se puede hablar de sobrecarga, ya que estos métodos no están definidos en **java.lang.Object**, con lo cual tenemos que definirlos. El método **writeObject** controla la información que es enviada al flujo de salida. Y **readObject**, lee la información escrita por **writeObject** .

La definición de **writeObject** ha de ser la siguiente:

```
private void writeObject (ObjectOutputStream s) throws IOException{
        s.defaultWriteObject();
        //código para escribir datos
}
```

El método **readObject** ha de leer todo lo que se ha escrito con **writeObject** en el mismo orden en el que se ha escrito. Además, puede realizar otras tareas necesarias para actualizar el estado del objeto.

```
private void readObject (ObjectInputStream s) throws IOException {
      s.defaultReadObject();
      //código para leer datos
      //actualización del estado del objeto, si es necesario
}
```

Para un control explícito del proceso de serialización, la clase ha de implementar la interfaz **Externalizable**. La clase es responsable de escribir y de leer su contenido, y ha de estar coordinada con sus clases base para hacer esto.

La definición de la interfaz **Externalizable** es la siguiente:

```
package java.io;

public interface Externalizable extends Serializable {
   public void writeExternal(ObjectOutput out) throws IOException;
   public void readExternal(ObjectOutput in) throws IOException,
      java.lang.ClassNotFoundException;
}
```

ANEXO

I. Cadenas delimitadas. StringTokenizer

Una **cadena delimitada** es aquella que está dividida en partes (*tokens*), las cuales se han formando teniendo en cuenta algún carácter o cadena delimitadora.

Por ejemplo la cadena "7647-34-123223-1-234" está delimitada por el guión "-", de forma que está formada por 5 tokens. Es muy común querer obtener cada **token** de la cadena para procesarla individualmente. Para ello, el paquete **java.util** dispone de la clase **StringTokenizer.**

Esta clase representa a una cadena delimitada, y además en cada momento hay un **puntero** interno que señala al siguiente **token** de la cadena. Con los métodos apropiados podremos avanzar por los tokens de la cadena.

• ¿Cómo construir una cadena delimitada?

Creando un objeto **StringTokenizer** pasándole dos parámetros: el **texto** delimitado y la **cadena delimitadora**.

Ejemplos:

```
StringTokenizer st1=new StringTokenizer("7647-34-123223-1-234","-");
StringTokenizer st2=new StringTokenizer("¿Dónde está mi gato?"," ");
```

El tokenizer **st1** tiene 5 tokens: "7647", "34", "123223", "1" y "234"
Y el tokenizer **st2** 4 tokens: "¿Dónde", "está", "mi", "gato?"

• Uso

Para obtener los **tokens** individuales de la cadena se usan estos métodos:

- ✓ **String nextToken().** Devuelve el siguiente token y provoca el avance del puntero una posición. La primera vez devuelve el primer texto de la cadena hasta que encuentra el delimitador, la segunda vez devuelve el siguiente texto delimitado y así sucesivamente. Si no hubiera más tokens devuelve la excepción **NoSuchElementException**. Por lo que conviene comprobar si hay más tokens.

- ✓ **boolean hasMoreTokens()**. Devuelve true si hay más tokens, false en caso contrario.

- ✓ **int countTokens().** Devuelve el número de tokens. El puntero no se mueve.

Ejemplo:

```
String cadena="10034-23-43423-1-3445";
StringTokenizer st=new StringTokenizer(cadena,"-");
```

```
//mostramos por pantalla todos los tokens
while (st.hasMoreTokens()){
    System.out.println(st.nextToken());
}
```

Ejemplos:

- **EjemploTokenizer.java:** programa que lee el contenido de un fichero de texto y lo analiza según un delimitador introducido por teclado.

- **Gastos.java:** fichero de texto llamado gastos.dat que contiene la siguiente información sobre el consumo mensual de una familia:

  Hipoteca 450.00
  Comida 300.00
  Viajes 100.00
  Ocio 150.00
  Muebles 150.00
  Otros 100.00

  Cada una de las líneas del fichero representa un concepto de gastos y la cantidad mensual asociada en €. Ambos valores están separados por un espacio en blanco. El programa lee dicha información y lo muestra por pantalla junto con el total de gatos acumulado ese mes con siguiente formato:

```
Hipoteca:  450.00 EUR
Comida:    300.00 EUR
Viajes:    100.00 EUR
Ocio:      150.00 EUR
Muebles:   150.00 EUR
Otros:     100.00 EUR

Total: 1250.00 EUR
```