

IES CHAN DO MONTE

C.S. de Desarrollo de Aplicaciones Multiplataforma

UNIDAD 1: Gestión de Ficheros

Índice

1. Manejo básico de ficheros en java	2
1.1 Concepto de fichero	2
1.2 Clasificación de los ficheros	2
1.2.1 Según el tipo de acceso a los datos:	3
1.3 Tipos de operaciones en ficheros	3
1.4 Paquete java.io	4
1.5 Stream (Flujos de datos)	5
1.6 Tipos de flujos en java	6
1.6.1 Flujos de bytes:	6
1.6.2 Flujos de caracteres	6
1.6.3 Utilización de los flujos	6
1.7 Los nombres de las clases de java.io	7
1.8 Jerarquía de los flujos de bytes	8
1.9 Jerarquía de los flujos de caracteres	9
1.10 Clases equivalentes entre flujos de bytes y flujos de caracteres	11
2. E/S estándar	11
2.1 Lectura de teclado en java	12
2.1.1 InputStream: el objeto System.in	12
2.1.2 Los Readers: InputStreamReader y BufferedReader	13
2.1.3 Convertir una cadena String a un tipo de datos numérico	14
2.1.4 La clase Scanner	15
2.2 Salida formateada: Clase PrintStream	17
3. Sistema de Ficheros y Directorios en Java	19
3.1 La clase FILE	19
3.1.1 Creación de un objeto File	19
3.1.2 Métodos de la clase File	19
4. Escribir y leer datos de archivos secuenciales binarios: Clases FileOutputStream y FileInputStream	22
5. Las clases orientadas al filtrado del flujo de bytes.	25
5.1 Acceso a datos primitivos: Clases DataInputStream y DataOutputStream	25
5.2 Flujos orientados a byte con buffer: Clases BufferedInputStream y BufferedOutputStream	28
5.3 Combinación de clases sobre los flujos de entrada y salida	29
6. Escribir y leer datos en archivos secuenciales de texto.	30
6.1 Clases FileReader y FileWriter	30
6.2 Codificación de caracteres	31
6.3 Las clases OutputStreamReader e InputStreamReader.	33
6.4 Buffer para el flujo de caracteres: Clases BufferedReader y BufferedWriter	34
7. Escritura formateada: la clase PrintWriter	35
8. La clase StreamTokenizer	36
9. Serialización	37
9.1 Interfaz Serializable	38
9.2 Excluir campos al serializar objetos	38
9.3 Flujos para la entrada y salida de objetos: ObjectInputStream e ObjectOutputStream	38
9.4 Escritura de objetos en ficheros	39
9.5 Lectura de objetos en ficheros	41
9.6 Serialización de objetos compuestos	42
10. Ficheros de acceso directo	43

1. Manejo básico de ficheros en java

Los programas usan variables para almacenar información: los datos de entrada, los resultados calculados y valores intermedios generados a lo largo del cálculo. Toda esta información es efímera, cuando salíamos del programa, todo lo que habíamos generado se pierde. A veces nos interesaría que la vida de los datos fuera más allá que la de los programas que los generaron. Es decir, que al salir de un programa, los datos generados quedaran guardados en algún lugar que permitiera su recuperación desde el mismo u otros programas. Por tanto, querríamos que dichos datos fueran **persistentes**.

Persistencia: Es la capacidad que tiene el programador para que sus datos se conserven al finalizar la ejecución de un proceso, de forma que se puedan recuperar y reutilizar en otros procesos.

Cuando se desea guardar información más allá del tiempo de ejecución de un programa, podemos optar por las siguientes posibilidades:

- Organizar esa información en uno o varios **ficheros** almacenados en algún soporte de almacenamiento persistente.
- Almacenar los datos en una **base de datos**.

En este capítulo veremos el uso básico de archivos en Java para conseguir persistencia de datos. Para ello, presentaremos conceptos básicos sobre archivos y algunas de las clases de la biblioteca estándar de Java para su creación y manipulación.

1.1 Concepto de fichero

Un archivo o fichero es una colección de datos homogéneos almacenados en un soporte físico del computador.

- **Datos homogéneos**: Almacena colecciones de datos del mismo tipo (igual que arrays/vectores)
- Cada elemento almacenado en un fichero se denomina **registro**, que se compone de campos.
- Puede ser almacenado en diversos soportes (disco duro, disquete, pendrive,...)

Desde el **punto de vista de más bajo nivel**, podemos definir un archivo (o fichero) como:

Un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica.

Es decir, un conjunto de 0s y 1s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD, entre otros.

1.2 Clasificación de los ficheros

Podemos utilizar varios criterios para distinguir diversas subcategorías de archivos. Estos tipos de archivos se diferenciarán desde el punto de vista de la programación, en que cada uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación. Según su contenido:

Sabemos que es diferente manipular números que Strings, aunque en el fondo ambos acaben siendo bits en la memoria del ordenador. Por eso, cuando manipulamos archivos, distinguiremos dos clases de archivos dependiendo del tipo de datos que contienen:

- **Los archivos de caracteres (o de texto)**
- **Los archivos de bytes (o binarios)**

Fichero binario			Fichero de texto		
0	00000000	Un número entero: 14	0	00110001	'1' (código ASCII 0x31)
1	00000000		1	00110100	'4' (código ASCII 0x34)
2	00000000		2	01101000	'h' (código ASCII 0x68)
3	00001110		3	01101111	'o' (código ASCII 0x6F)
4	00000000	Otro número entero: 33	4	01101100	'l' (código ASCII 0x6C)
5	00000000		5	01100001	'a' (código ASCII 0x61)
6	00000000		
7	00100001				
...	...				

- Un **fichero de texto** es aquél formado exclusivamente **por caracteres** y que, por tanto, puede crearse y visualizarse usando un editor. Las operaciones de lectura y escritura trabajarán con caracteres.

Por ejemplo, los ficheros con código java son ficheros de texto.

- En cambio un **fichero binario** ya no está formado por caracteres sino que los bytes que contiene pueden representar otras cosas como números, imágenes, sonido, etc.

NOTA: Para “entender” los contenidos de un fichero es necesario conocer de antemano el tipo de datos que contiene.

1.2.1 Según el tipo de acceso a los datos:

Existen dos modos básicos de acceso a la información contenida en un archivo:

- **Secuencial**
- **Acceso aleatorio**

- **Acceso Secuencial:** En este caso, **los datos son leídos secuencialmente**, desde el principio hasta el final. La información del archivo es una secuencia de bytes (o caracteres) de manera que para acceder al byte (o carácter) N se ha de haber accedido anteriormente a los N-1 anteriores.
- **Acceso aleatorio (“Random” o directo):** los archivos de acceso aleatorio permiten acceder a los datos en forma no secuencial, desordenada. Nos **permite acceder directamente a la información del byte N**.

En java podemos incluir otro tipo de acceso a los datos:

- **Concatenación (tuberías o pipes):** Muchas veces es útil **realizar conexiones entre programas** que corren simultáneamente dentro de una misma máquina, de forma que lo que uno produce se envía por un “tubo” para ser recibido por el otro, que está esperando a la salida del tubo (sin tener que utilizar una memoria intermedia). Las tuberías o pipes cumplen esta función.

1.3 Tipos de operaciones en ficheros

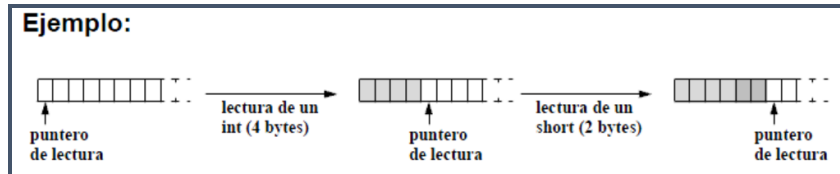
Las operaciones más comunes en ficheros son:

- Operación de Creación
- Operación de Apertura. Varios modos:
 - Sólo lectura
 - Sólo escritura
 - Lectura y Escritura
- Operaciones de lectura / escritura
- Operaciones de inserción / borrado
- Operaciones de renombrado / eliminación
- Operación de desplazamiento dentro de un fichero
- Operación de ordenación
- Operación de cierre

Punteros de lectura y escritura

Las **operaciones sobre fichero se van a realizar en el byte** que señalen **los punteros de lectura y escritura**:

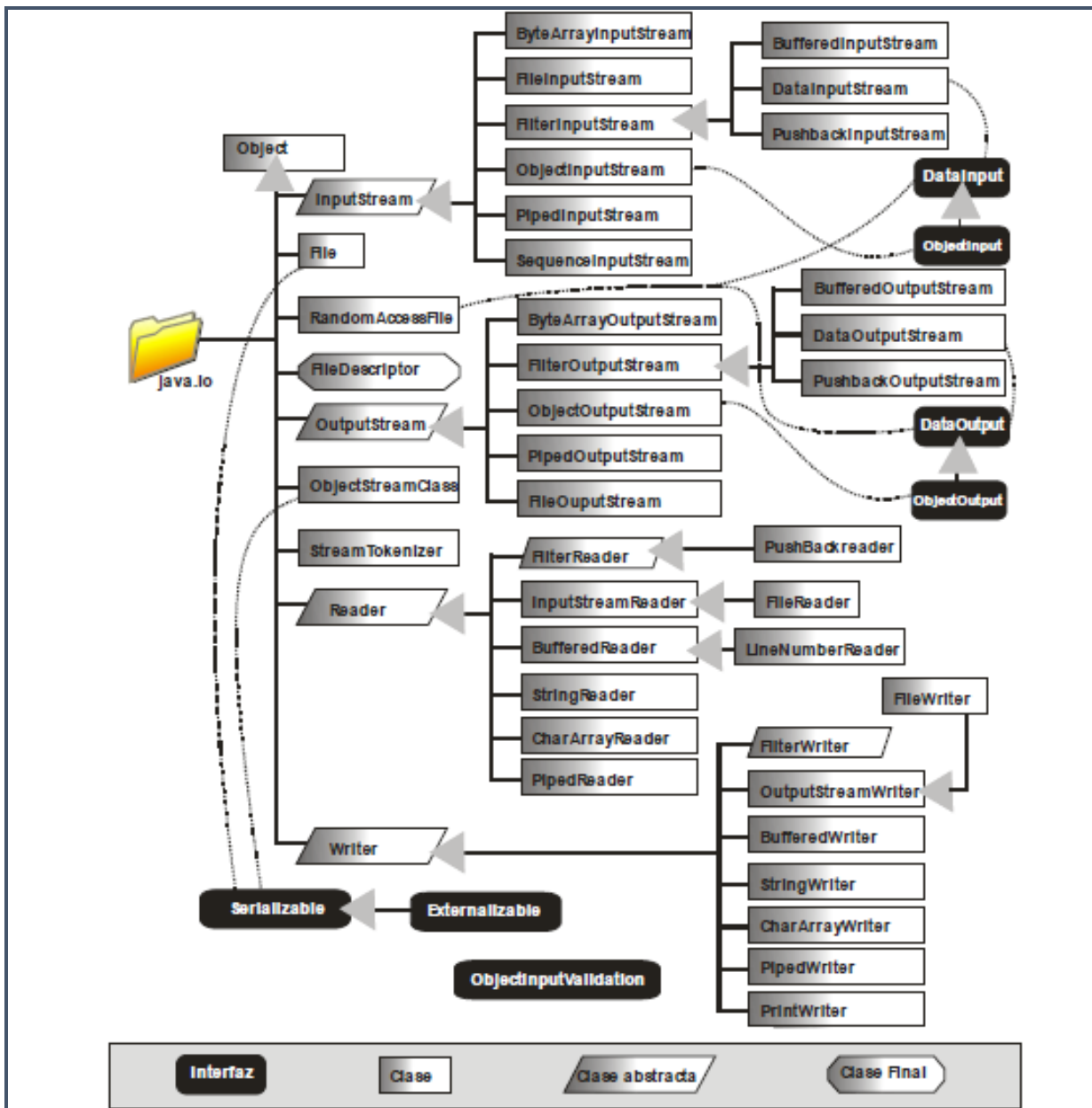
- Indican el próximo byte a leer o a escribir
- Gestionados automáticamente por el sistema operativo
- Cuando se abren, se comienzan apuntando al primer byte del fichero
- Van avanzando por el fichero según se van leyendo sus contenidos o escribiendo datos.



1.4 Paquete java.io

El paquete **java.io** contiene todas las **clases relacionadas con las funciones de entrada (input) y salida (output)**.

Las clases que contienen el paquete java.io se muestran en la figura siguiente:



RECORDATORIO:

Clase final: se declara como la clase que termina una cadena de herencia. No se puede heredar de una clase final. Por ejemplo, la clase Math es una clase final.

Clase abstracta: Una clase abstracta es una clase de la que no se puede crear objetos. La utilidad de estas clases estriba en que otras clases hereden de ésta, por lo que con ello conseguiremos reutilizar código. Para declarar una clase como abstracta utilizamos la palabra clave abstract. Los métodos para los que no aporte una implementación serán declarados a su vez abstractos. Si una clase tiene un método abstract es obligatorio que la clase sea abstract. Todas las subclases que hereden de una clase abstracta tendrán que redefinir los métodos abstractos dándoles una implementación

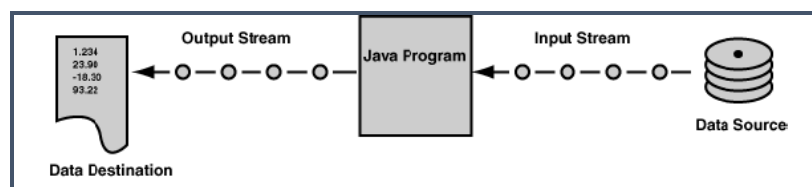
Interfaz: es una clase abstracta pura, es decir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código) obligando a que ciertas clases utilicen estos mismos métodos (nombres y parámetros).

1.5 Stream (Flujos de datos)

Java se basa en las secuencias de datos para dar facilidades de entrada y salida. Una secuencia es una corriente de datos en serie entre un emisor y un receptor de datos en cada extremo.

Los programas en Java realizan la E/S a través de **streams (Flujos)**, es decir, cualquier programa realizado en Java que necesite llevar a cabo una operación de E/S lo hará a través de un stream.

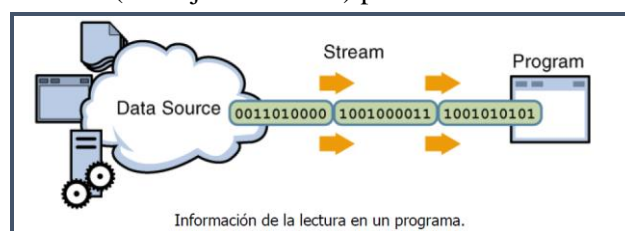
Un stream, cuya traducción literal es "flujo", es una abstracción de todo aquello que produzca o consuma información



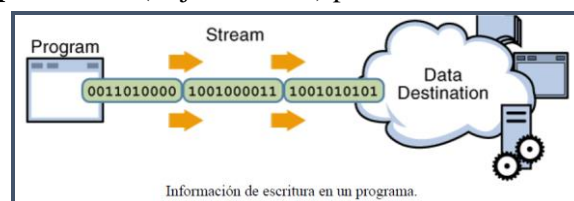
Un flujo no está relacionado con un dispositivo físico, comportándose todos los flujos de la misma manera aunque traten con dispositivos distintos. Debido a esta característica **se pueden aplicar las mismas clases a cualquier tipo de dispositivo**.

La **vinculación del stream al dispositivo físico lo lleva a cabo el sistema de entrada y salida de Java**. Las clases y métodos de I/O que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando, luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de ficheros o un socket de red liberando a nuestro código de tener que saber con quién está interactuando.

Un programa utiliza **input stream** (un flujo de entrada) para obtener los datos leídos de una fuente:



Un programa utiliza un **output stream** (flujo de salida) para escribir los datos a un destino:



La fuente de datos (data source) y el destino de los datos (data destination) de los diagramas anteriores puede ser cualquier cosa que genera o consume datos. Esto incluye obviamente archivos de disco, pero una fuente o un destino puede ser otro programa, un dispositivo (impresora, periférico, etc.), un socket de red (es un método para la comunicación entre un programa del cliente y un programa del servidor en una red) o un array.

1.6 Tipos de flujos en java

Podemos distinguir básicamente dos tipos de flujos:

- Los **flujos de bytes**: Los datos fluyen en serie, byte a byte.
- Los **flujos de caracteres**: Transmiten caracteres Java

1.6.1 Flujos de bytes:

Nos proporciona un medio adecuado para el manejo de **entradas y salidas de bytes** y su uso lógicamente está orientado a la *lectura y escritura de datos binarios*.

El tratamiento del flujo de bytes viene gobernado por dos clases abstractas: **InputStream** y **OutputStream**.

Cada una de estas clases abstractas tiene varias subclases concretas que controlan las diferencias ente los distintos dispositivos de I/O que se pueden utilizar. Así mismo, estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan **read()** y **write()** que leen y escriben bytes de datos respectivamente.

1.6.2 Flujos de caracteres

Proporciona un medio conveniente para el manejo de *entradas y salidas de caracteres*.

Dichos flujos **usan codificación Unicode** y, por tanto, se pueden internacionalizar.

Este es un modo que Java nos proporciona para manejar caracteres pero al nivel más bajo todas las operaciones de I/O son orientadas a byte.

Al igual que la anterior el flujo de caracteres también viene gobernado por dos clases abstractas: **Reader** y **Writer**. Dichas clases *manejan flujos de caracteres Unicode*. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos **read()** y **write()** que, en este caso, leen y escriben caracteres de datos respectivamente.

1.6.3 Utilización de los flujos

La utilización de los flujos es un nivel de abstracción que hace que un programa no tenga que saber nada del dispositivo, lo que se traduce en una facilidad más a la hora de escribir programas, ya que los algoritmos para leer y escribir datos serán siempre más o menos los mismos. En general, las operaciones serán:

■ Lectura

1. **Abrir un flujo a una fuente de datos** (creación del objeto stream): Teclado, Fichero, Socket

Nota: Los sockets son un sistema de comunicación entre procesos de diferentes máquinas de una red. Mas exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información.

2. **Mientras existan datos disponibles**

- Leer datos

3. **Cerrar el flujo** (método close)

■ Escritura

1. **Abrir un flujo** a una fuente de datos (creación del objeto stream): Pantalla, Fichero, Socket
2. **Mientras existan datos disponibles**
 - Escribir datos
3. **Cerrar el flujo** (método close)

Nota: Para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos.

Un fallo en cualquier punto produce la excepción **IOException**

1.7 Los nombres de las clases de java.io

Las clases de **java.io** siguen una nomenclatura sistemática 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 y directorios
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubería de datos
Buffered	Buffer
Filter	Filtro o procesos sobre el stream
Data	Intercambio de datos en formato propio de Java (datos primitivos)
Object	Persistencia de objetos
Print	Imprimir

Un buffer es un espacio de memoria intermedia. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.

■ Clases que indican origen/destino de los datos.

- Clases de E/S en disco:
 - *FileReader* - *FileWriter*
 - *FileInputStream* - *FileOutputStream*.
- Clases de E/S en memoria:
 - *StringReader* *StringWriter*
 - *CharArrayReader* *CharArrayWriter*
 - *ByteArrayInputStream* *ByteArrayOutputStream*
 - *StringBufferInputStream*.
- Conexión bilateral para transmisión de datos (tuberías):
 - *PipedReader* *PipedWriter*
 - *PipedInputStream* *PipedOutputStream*

■ Clases que modifican comportamiento.

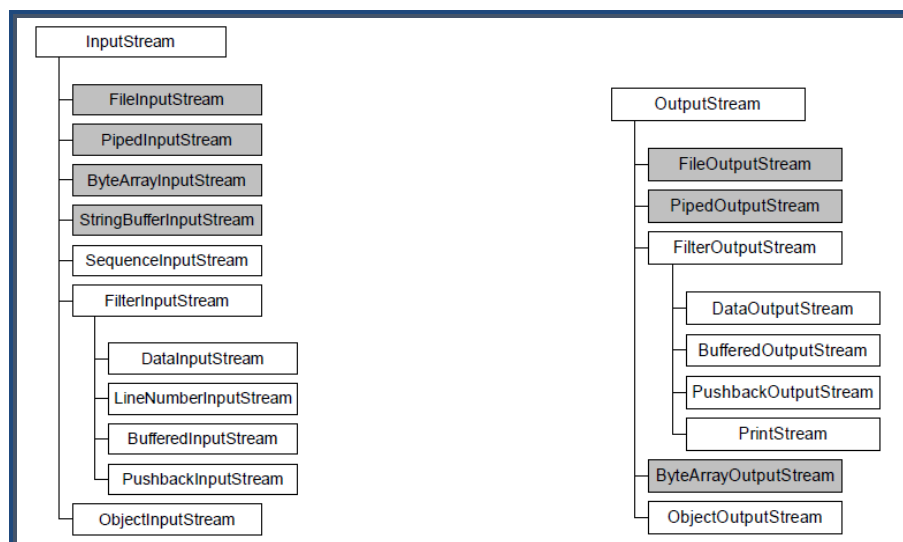
- Utilización de un buffer minimizando el acceso al dispositivo:
 - *BufferedReader* *BufferedWriter*
 - *BufferedInputStream* *BufferedOutputStream*
- Conversión de un flujo de byte en uno de caracteres. Es la conexión con la jerarquía
 - *InputStream/OutputStream*
 - *InputStreamReader* *OutputStreamWriter*
- Serialización de objetos
 - *ObjectInputStream* y *ObjectOutputStream*
- Filtros o procesos sobre el stream
 - *FilterReader* *FilterWriter* *FilterInputStream* *FilterOutputStream*
- Manejar datos en formato de Java (independencia de la plataforma)
 - *DataInputStream* *DataOutputStream*
- Con métodos adaptados para imprimir tipos de Java
 - *PrintWriter* o *PrintStream*

- Representación de la información
 - Flujos de bytes: clases *InputStream* y *OutputStream*
 - Flujos de caracteres: clases *Reader* y *Writer*
 - Se puede pasar de un flujo de bytes a uno de caracteres con *InputStreamReader* y *OutputStreamWriter*
- Propósito
 - Entrada: *InputStream*, *Reader*
 - Salida: *OutputStream*, *Writer*
 - Lectura/Escritura: *RandomAccessFile*
 - Transformación de los datos
 - Realizan algún tipo de procesamiento sobre los datos (p.e. *buffering*, conversiones, filtrados): *BufferedReader*, *BufferedWriter*
- Acceso
 - Secuencial
 - Aleatorio - (*RandomAccessFile*)

1.8 Jerarquía de los flujos de bytes

- Los programas **utilizan secuencias de bytes** para realizar la **entrada y salida de bytes (8-bits)**.
- Todas las **clases de flujos de bytes** **descienden** de las clases abstractas ***InputStream* y *OutputStream***.
- Al tratarse de clases abstractas, **no vamos a poder crear objetos de estas clases** porque su funcionalidad está "incompleta" (tienen métodos que no están definidos, implementándolos las subclases), y así nos permiten representar un flujo de datos de entrada o salida binario cualquiera.
- Cada una de las clases hijas de *InputStream* y *OutputStream* **proporciona** una **funcionalidad más específica a la clase padre**, por lo tanto, **tendremos que crear objetos de alguna de sus subclases**.
- En la siguiente figura podemos ver la jerarquía de los flujos de entrada y salida de bytes de Java:

Las clases ***InputStream* y *OutputStream*** son clases abstractas de las que se derivan las clases que permiten **crear flujos de bytes de 8 bits** de lectura e escritura.



Métodos de InputStream

int	<code>available()</code> Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	<code>close()</code> Cierra la corriente de entrada y libera los recursos del sistema que esté usando.
void	<code>mark(int readlimit)</code> Marca la posición actual de la corriente de entrada.
boolean	<code>markSupported()</code> Prueba si esta corriente de entrada soporta los métodos <code>mark</code> y <code>reset</code> .
abstract int	<code>read()</code> Lee el siguiente byte de la corriente de entrada.

<code>int</code>	<code>read(byte[] b)</code> Lee bytes de la corriente de entrada y los deposita en el vector <code>b</code> .
<code>int</code>	<code>read(byte[] b, int off, int len)</code> Lee hasta <code>len</code> bytes de la corriente de entrada y los deposita en <code>b</code> a partir de <code>off</code> .
<code>void</code>	<code>reset()</code> Coloca la corriente de entrada en la posición que tenía la última vez que se invocó <code>mark</code> .
<code>long</code>	<code>skip(long n)</code> Salta y descarta los siguientes <code>n</code> bytes de la corriente de entrada.

Métodos de OutputStream	
<code>void</code>	<code>close()</code> Cierra la corriente de salida y libera los recursos del sistema que está utilizando.
<code>void</code>	<code>flush()</code> Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
<code>void</code>	<code>write(byte[] b)</code> Escribe todos los <code>b.length</code> bytes del vector <code>b</code> en la corriente de salida.
<code>void</code>	<code>write(byte[] b, int off, int len)</code> Escribe <code>len</code> bytes del vector <code>b</code> comenzando en la posición <code>off</code> .
<code>abstract void</code>	<code>write(int b)</code> Escribe un byte especificado por el <code>int b</code> .

Los **métodos más importantes** son los que sirven para **leer y escribir: read y write**.

Los tres métodos `read` sirven para leer bytes:

```
public abstract int read() throws IOException;
public int read(byte[] b) throws IOException;
public int read(byte[] b, int pos, int n) throws IOException;
```

El primero lee un solo byte y devuelve su valor como un entero entre 0 y 255. El segundo lee tantos bytes como pueda hasta llenar el vector (array) `b` que se le pasa como parámetro o hasta que se acabe la corriente. El último lee los `n` bytes y los pone en el vector `b`, a partir de la posición `pos`. ¿Qué ocurre si la corriente de entrada se termina? En ese caso el resultado de los métodos `read` es -1.

Los tres métodos de escritura `write` son análogos a los de lectura.

```
public abstract void write(int b) throws IOException;
public void write(byte[] b) throws IOException;
public void write(byte[] b, int pos, int n) throws IOException;
```

El primero escribe un byte aunque se le pase como parámetro un entero. El método convierte el entero a byte y luego lo escribe. Para no perder información al método `write(int n)` sólo se le deben pasar valores `n` entre 0 y 255. El segundo método `write` escribe todos los bytes del vector `b` que se le pasa como parámetro y el tercero escribe `n` bytes del vector `b` comenzando desde la posición `pos`.

1.9 Jerarquía de los flujos de caracteres

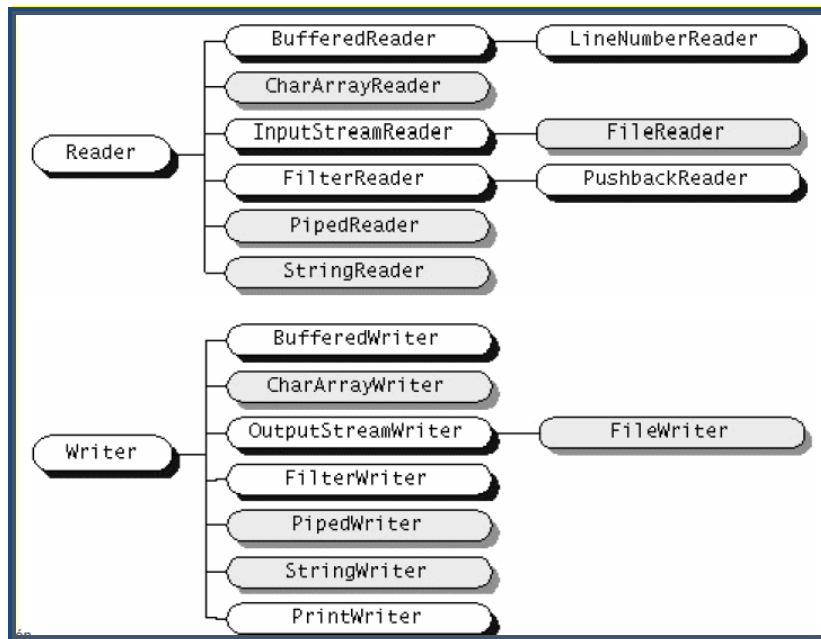
Existen dos variaciones para las Clases `InputStream` y `OutputStream` que son: **Writer y Reader**.

La principal diferencia entre estas clases es que las primeras ofrecen lo que es conocido como byte-orientated I/O, mientras `Writer` y `Reader` ofrecen character-based I/O, en otras palabras las clases `InputStream` y `OutputStream` solamente soportan "Streams" de 8-bits byte, mientras las **Clases Writer y Reader soportan "Streams" de 16-bits**.

La importancia de 16-bits radica en Java que utiliza Unicode, al utilizarse 8-bits no es posible emplear muchos caracteres disponibles en Unicode, además debido a que las Clases `Writer` y `Reader` son una adición más reciente al JDK, estas poseen mayor velocidad de ejecución a diferencia de sus contrapartes `InputStream` y `OutputStream`.

Las clases **Reader y Writer** son clases abstractas de las que se derivan las clases que permiten **crear flujos de caracteres de 16 bits (Unicode)** de lectura e escritura.

La jerarquía de clases que parte tanto de la clase `Reader` como `Writer` se puede ver en la siguiente imagen:



Metodos de Reader	
abstract void	<code>close()</code> Cierra la corriente de entrada y libera los recursos del sistema que esté usando.
void	<code>mark(int readlimit)</code> Marca la posición actual de la corriente de entrada.
boolean	<code>markSupported()</code> Prueba si esta corriente de entrada soporta los métodos <code>mark</code> y <code>reset</code> .
int	<code>read()</code> Lee un character.
int	<code>read(char[] cbuf)</code> Lee caracteres de la corriente de entrada y los deposita en el vector de caracteres <code>cbuf</code>
abstract int	<code>read(char[] cbuf, int off, int len)</code> Lee <code>len</code> caracteres de la corriente de entrada y los deposita en el vector de caracteres <code>cbuf</code> a partir de <code>off</code>
boolean	<code>ready()</code> Indica si el flujo de entrada está listo para ser leído
void	<code>reset()</code> Restablece el flujo de entrada.
long	<code>skip(long n)</code> Salta <code>n</code> caracteres.

Métodos de Writer	
Writer	<code>append(char c)</code> Añade el carácter especificado en el flujo de entrada
abstract void	<code>close()</code> cierra el flujo de salida
abstract void	<code>flush()</code> Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
void	<code>write(char[] cbuf)</code> Escribe un array de caracteres
abstract void	<code>write(char[] cbuf, int off, int len)</code> Escribe <code>len</code> caracteres a partir del desplazamiento marcado por <code>off</code> .
void	<code>write(int c)</code> Escribe un carácter
void	<code>write(String str)</code> Escribe una cadena
void	<code>write(String str, int off, int len)</code> Escribe una porción de la cadena empezando en <code>off</code> y de longitud <code>len</code> .

1.10 Clases equivalentes entre flujos de bytes y flujos de caracteres

	Clases de <i>streams</i> de caracteres	Clases de <i>streams</i> de bytes equivalentes
Entrada	Reader BufferedReader LineNumberReader CharArrayReader InputStreamReader FileReader FilterReader PushbackReader PipedReader StringReader	InputStream BufferedInputStream LineNumberInputStream ByteArrayInputStream (ninguna) FileInputStream FilterInputStream PushbackInputStream PipedInputStream StringBufferInputStream
Salida	Writer BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PrintWriter PipedWriter StringWriter	OutputStream BufferedOutputStream ByteArrayOutputStream FilterOutputStream (ninguna) FileOutputStream PrintStream PipedOutputStream (ninguna)

2. E/S estándar

Todos los programas Java importan el paquete **Java.lang** automáticamente. Este paquete define una clase llamada **System**:

```

java.lang
Class System

java.lang.Object
  java.lang.System

public final class System
  extends Object
  
```

Es una clase final y todos sus contenidos son privados.

- No se puede instanciar un objeto de esa clase.
- La clase System siempre está ahí disponible para que se pueda invocar cualquiera de sus métodos.
- Todos los miembros del paquete java.lang se pueden usar directamente, **no** hay que importarlos.

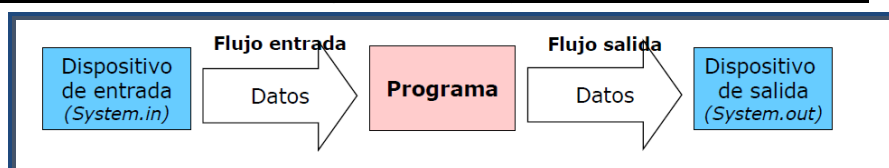
- Contiene tres variables **con flujos predefinidos** llamadas **in**, **out** y **err**, que se pueden utilizar sin tener una referencia a un objeto System específico.

Fields	
Modifier and Type	Field and Description
static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

En Java se accede a la E/S estándar a través de los atributos estáticos de la clase **java.lang.System**

- **System.in**: implementa la entrada estándar. Por defecto es el teclado.
- **System.out**: implementa la salida estándar. Por defecto es la pantalla.
- **System.err**: implementa la salida de error. Por defecto es la pantalla

Estos flujos **standard** el sistema se encarga de abrirlos y cerrarlos automáticamente.



- **System.in**:
 - Instancia de la clase **InputStream**: **flujo de bytes** de entrada

- Métodos:
 - read()**: permite leer **un byte** de la entrada como entero
 - skip(n)**: ignora n bytes de la entrada
 - available()**: número de bytes disponibles para leer en la entrada

■ System.out:

- Instancia de la clase **PrintStream** (subclase de **OutputStream**): **flujo de bytes** de salida
- Metodos para impresión de datos
 - print()**, **println()**
 - flush()**: vacía el buffer de salida escribiendo su contenido

■ System.err

- Funcionamiento similar a System.out
- Se utiliza para enviar mensajes de error (por ejemplo a un fichero de log o a la consola)

Ejemplo: **Leer caracteres por teclado (byte a byte) hasta pulsar retorno de carro formado una cadena.** Se imprime la cadena formada y el número total de bytes.

```
package ejentradastandardbyte;
import java.io.*;
public class EjEntradaStandardbyte {
    //se tiene que capturar la excepción IOException
    public static void main(String[] args) throws IOException {
        int c, contador = 0;
        String cadena= new String();
        // se lee byte a byte hasta encontrar el fin de línea
        while( (c = System.in.read() ) != '\n' )
        {
            contador++;
            //Concatenamos el caracter a cadena
            cadena+=(char) c;
        }
        System.out.println("cadena introducida:"+cadena);
        System.out.println( "Contados " + contador + " bytes en total." );
    }
}
```

2.1 Lectura de teclado en java

2.1.1 InputStream: el objeto System.in

En java tenemos accesible el teclado desde **System.in**, que es un **InputStream** del que podemos leer bytes.

System.in es un objeto de la clase InputStream.

Para java, un **InputStream** es cualquier cosa de la que se leen bytes. Puede ser el teclado, un fichero, un socket, o cualquier otro dispositivo de entrada. Esto, por un lado es una ventaja. Si todas esas cosas son **InputStream**, podemos hacer código que lea de ellas sin saber qué estamos leyendo.

Por ejemplo, podemos leer bytes del teclado de esta forma:

```
// Lectura de un byte
int mybyte = System.in.read();
```

Cómo un **InputStream** es para leer bytes, sólo tiene métodos para leer bytes. El problema de leer bytes, es que luego debemos convertirlos a lo que necesitemos. Por ejemplo, si tecleamos una letra A mayúscula, el byte leído es el 65, correspondiente a la A mayúscula en código ASCII. Si tecleamos un 3 y un 2, es decir, un 32, leeremos dos bytes 51 y 52, correspondientes a los caracteres ASCII del 3 y del 2, NO leeremos un 32.

2.1.2 Los Readers: InputStreamReader y BufferedReader

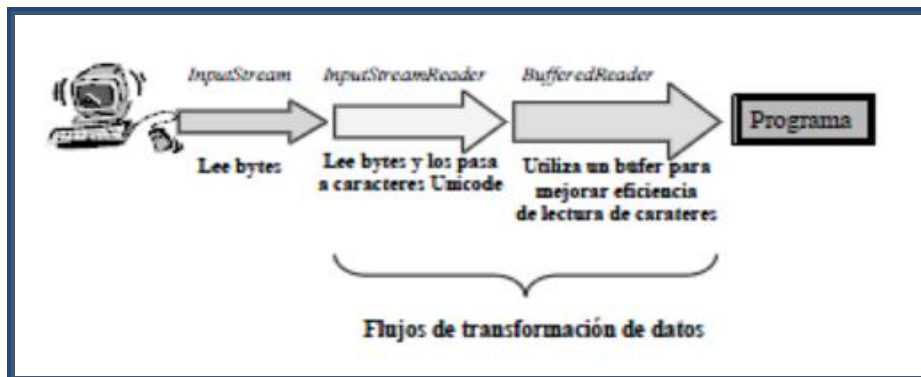
Para java, una **clase Reader es una clase que lee caracteres**. Un Reader tiene métodos para leer caracteres. Con esta clase ya podríamos trabajar. La pena es que seguimos teniendo System.in, que es un InputStream y no un Reader.

¿Cómo convertimos el System.in en Reader?: Hay una clase en java, **InputStreamReader**, que nos hace esta conversión. Para obtener un Reader, únicamente tenemos que instanciar un InputStreamReader pasándole en el constructor un InputStream. El código es el siguiente:

```
InputStreamReader entrada = new InputStreamReader(System.in);
```

Estamos declarando una variable "entrada" de tipo InputStreamReader. Creamos un objeto de esta clase haciendo new InputStreamReader(...). Entre paréntesis le pasamos el InputStream que queremos convertir a Reader, en este caso, el System.in.

Hasta este momento solo se puede leer carácter a carácter, ahora bien, si quisiéramos leer, por ejemplo, 10 caracteres del teclado o hasta que se pulse la tecla intro, si sólo usamos InputStreamReader, como lee caracteres sueltos, tendríamos que decirle cuántos queremos (que puede que no lo sepamos), o bien ir pidiendo de uno en uno hasta que no haya más. Leer carácter a carácter puede llegar a seguir incómodo, sería mejor leer de una sola vez un conjunto de caracteres. En Java existe una clase que **nos permite leer una línea completa**: la clase **BufferedReader**.



El mecanismo para obtener un objeto BufferedReader es a partir de otro Reader cualquiera (por ejemplo, el InputStreamReader), por tanto, se instancia pasándole en el constructor un objeto de tipo Reader. El código sería:

```
InputStreamReader entrada = new InputStreamReader(System.in);
BufferedReader entradabuffer = new BufferedReader(entrada);

O simplificándolo:

BufferedReader entradabuffer = new BufferedReader
(new InputStreamReader(System.in));
```

Se crea un InputStreamReader a partir de System.in y pasamos el InputStreamReader al constructor de BufferedReader, para que la lectura que se haga sobre el objeto creado en ese momento sea en realidad realizadas sobre System.in, para que al final el resultado sea leer una línea completa.

Para utilizar el método BufferedReader debemos **importar los paquetes: BufferedReader, InputStreamReader, IOException**.

Esto es porque necesitamos crear un objeto del tipo BufferedReader el cual recibe como parámetro un objeto de la clase InputStreamReader. El uso del BufferedReader requiere forzosamente del manejo de excepciones por ello también es necesario importar la clase IOException

Para realizar la captura desde el teclado, **BufferedReader cuenta con el siguiente método**:

```
readLine

public String readLine()
                throws IOException
```

- **readLine():** Lee una línea de texto hasta que encuentra un carácter de salto de línea (\n) y retorno de carro (\r), o sea hasta que pulsemos enter. Se obtiene un string pero si se quiere manipular como otro tipo de dato se tendrá que hacer una conversión.

```

BufferedReader entradabuffer = new BufferedReader
                                (new InputStreamReader(System.in));

String texto;
try{
    texto=entradabuffer.readLine();
}
catch (IOException ioe){
    ioe.printStackTrace();
    /* llama a printStackTrace() de la excepción que se ha provocado. Esta llamada escribe el
    texto de error producido, y además nos dice exactamente en qué línea de código se produce */
}

```

2.1.3 Convertir una cadena String a un tipo de datos numérico

Si se quiere leer un número del teclado, el usuario escribe, por ejemplo 123, con la clase **BufferedReader** obtendremos un **String** que contiene "123", es decir, tres caracteres. Si lo que se pretende es leer un número, entonces tendremos que hacer una conversión del **String** al tipo numérico deseado.

- Convertir un **String** a un valor entero.

Para convertir un **String** (cadena) a un valor numérico (**int**) hay que emplear el método estático de la **clase Integer**, **parseInt**

```

parseInt

public static int parseInt(String s)
                        throws NumberFormatException

```

Para convertir el **String** a **int** se necesita que el **String** sea exactamente un **int**. Cualquier carácter o letra que tenga el **String** que no sea válida, se hará que la conversión falle.

Ejemplo:

```

int entero;
try{
    entero=Integer.parseInt(texto);
}
catch (java.lang.NumberFormatException e){
    System.out.println("Error, no se puede convertir, el numero no es un entero");
}

```

- Convertir un **String** a un valor entero largo.

Para convertir un **String** (cadena) a un valor numérico (**long**) hay que emplear el método estático de la **clase Long**, **parseLong**

```

parseLong

public static long parseLong(String s)
                        throws NumberFormatException

```

- Convertir un **String** a un valor entero corto.

Para convertir un **String** (cadena) a un valor numérico (**short**) hay que emplear el método estático de la **clase Short**, **parseShort**

```

parseShort

public static short parseShort(String s)
                        throws NumberFormatException

```

- Convertir un **String** a un valor **Float**.

Para convertir un **String** (cadena) a valor decimal(**float**), hay que emplear el método estático de la **clase Float**, **parseFloat**

```

parseFloat

public static float parseFloat(String s)
                        throws NumberFormatException

```


- Convertir un String a un valor Double

Para convertir un String (cadena) a un valor decimal (Double), hay que emplear el método estático de la **clase Double**, **parseDouble**

```
parseDouble

public static double parseDouble(String s)
    throws NumberFormatException
```

2.1.4 La clase Scanner

Esta clase que se encuentra disponible desde Java 1.5, nos permite leer datos de una forma más sencilla que el clásico InputStream con un BufferedReader.

La clase Scanner tiene varios constructores que admiten, además de System.in, cosas como secuencias de bytes o ficheros.

El **constructor que se utiliza par leer de la consola** es:

```
Scanner

public Scanner(InputStream source)
```

Para utilizar Scanner para leer datos del teclado, tan solo tenemos que crearnos un objeto de tipo Scanner (importando previamente el paquete **java.util.Scanner**) e indicándole a este que lea de la consola con **System.in**. Nos quedaría lo siguiente:

```
Scanner pruebaScanner = new Scanner(System.in);
```

Algunos métodos de la clase Scanner para lectura de datos:

String	next() lee una cadena, no permite espacios en blanco
boolean	nextBoolean() lee una valor boolean
byte	nextByte() lee un byte
double	nextDouble() lee un valor decimal double
float	nextFloat() lee un valor decimal float.
int	nextInt() lee un valor int.
String	nextLine() lee una cadena hasta que se pulse enter, permitiendo espacios en blanco.
long	nextLong() lee un valor numérico long.
short	nextShort() lee un valor numérico short.

```
package ejtecladoscanner;
import java.util.Scanner;
public class EjTecladoScanner {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("¿Cómo te llamas? ");
        String nombre = sc.nextLine(); // leer una cadena de caracteres
        System.out.println("Es un placer conocerte "+nombre);
        System.out.print("¿Que edad tienes? ");
        int edad = sc.nextInt(); // leer una cadena de caracteres
        System.out.print("¿pues aparentas menos edad? ");

    }
}
```

Otros métodos de Scanner son:

void	<code>close()</code> Cierra el Scanner
boolean	<code>hasNext()</code> Indica si quedan mas cadenas por leer
boolean	<code>hasNextBoolean()</code> Indica si es posible leer mas datos que se interpreten como un boolean
boolean	<code>hasNextByte()</code> Indica si es posible leer mas datos que se interpreten como un byte
boolean	<code>hasNextDouble()</code> Indica si es posible leer mas datos que se interpreten como un double
boolean	<code>hasNextFloat()</code> Indica si es posible leer más datos que se interpreten como un booleano float.
boolean	<code>hasNextInt()</code> Indica si es posible leer más datos que se interpreten como un booleano int.
boolean	<code>hasNextLine()</code> Indica si quedan más cadenas por leer.
boolean	<code>hasNextLong()</code> Indica si es posible leer más datos que se interpreten como un booleano long.
boolean	<code>hasNextShort()</code> Indica si es posible leer más datos que se interpreten como un booleano short.
Scanner	<code>useDelimiter(String pattern)</code> cambia los delimitadores que van a separar los items

Excepciones que pueden lanzar los métodos de Scanner:

- **NoSuchElementException**: no quedan más cadenas.
- **IllegalStateException**: el scanner está cerrado.
- **InputMismatchException**: el dato leído no es del tipo esperado.

- **Uso de la clase Scanner para leer de Fichero.** Podemos utilizar el siguiente constructor:

```
Scanner

public Scanner(File source)
    throws FileNotFoundException
```

-Para leer proveniente de un String

```
Scanner

public Scanner(String source)
```

Ej: Tenemos un fichero de texto con la siguiente información que corresponde a identificativo, nombre y edad de una persona:

Ejemplo: Bloc de notas

Archivo Edición Formato

```
1, Antonio, 4
2, Luis, 7
4, Anton, 8
```

Vamos a hacer un pequeño programa en java usando Scanner que nos permita leer estos datos del fichero y visualizar esta información por la pantalla.

```
package ejficheroscanner;
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Scanner;
public class EJFicheroScanner {
    public static void main(String[] args) {

        File f = new File("C:/FicherosJava/ejemplo1.txt");
        Scanner s;
        try {
            s = new Scanner(f);
            while (s.hasNextLine()) {
                String linea = s.nextLine();
                Scanner sl = new Scanner(linea);
                //patron de búsqueda: uno o mas espacio, seguido de coma y de uno o más espacio
                sl.useDelimiter("\\s*,\\s*");
                System.out.print(sl.next()+" ", " ");
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```

        System.out.print(sl.next() + ", ");
        System.out.println(sl.next());
    }
    s.close();
}
catch (FileNotFoundException e) {
    System.out.println("fichero no encontrado");
}
}
}

```

La **clase Scanner admite expresiones regulares** como patrones de búsqueda, por lo que podemos leer trozos de línea directamente usando los separadores que queramos o buscando expresiones concretas.

Por ejemplo, si pedimos por teclado una fecha dd/mm/yy, necesitamos comprobar si la cadena leída cumple ese patrón: dos cifras, una barra, dos cifras, otra barra y otras dos cifras. Una expresión regular que se ajusta al anterior patrón podría ser:

`\\d\\d\\d\\d\\d\\d` donde `\\d` quiere decir dígito, como es un carácter comodín y no forma parte de la cadena de búsqueda, hay que escaparlo con otra `\\`.

Si por ejemplo, se lee una fecha con un solo dígito como 1/1/12, sólo tiene un dígito en cada caso, así que no cumple el patrón y devuelve false.

Si queremos algo más sofisticado, para que se admitan los días y meses con uno o dos dígitos, podemos usar el siguiente patrón:

`\\d{1,2}\\d{1,2}\\d{1,2}`. Ahora, la expresión `\\d{1,2}` indica un dígito entre 1 y 2 veces, es decir, uno o dos dígitos.

Hay muchas opciones, solo vamos a nombrar algunas, como por ejemplo:

- El operador `*` representa que el patrón indicado debe aparecer 0 o mas veces.
- El operador `+` representa que el patrón indicado debe aparecer 1 o más veces.
- El carácter `"?"`: Indica que el símbolo que le precede puede aparecer una vez o ninguna. Ejemplo `"H?ola"` describe a `Hola` y a `ola`.
- El carácter `"^"`: Representa el inicio de una cadena, de la forma que si ponemos un ejemplo con este carácter y otros entre paréntesis buscará las cadenas con esos caracteres de inicio. Cuando se emplea dentro de los corchetes la búsqueda muestra aquellas cadenas que no tienen esos caracteres al inicio.
- El carácter `"$"`: Representa el final de una cadena o el final de línea, es muy útil para avanzar entre párrafos.
- `[]` agrupar caracteres en grupos o clases.
- `|` Sirve para indicar una de varias opciones
- `\\s` para localizar caracteres como espacios, tabuladores `\\.S` para lo contrario a `\\s`.
- `\\d` para localizar cadenas con un digito. `\\.D` para lo contrario al `\\d`.
- `\\A` para empezar la búsqueda por el principio de la cadena.
- `\\Z` para empezar la búsqueda por el final de la cadena.

2.2 Salida formateada: Clase PrintStream

Salida por consola:

`System.out` y `System.err` **se definen como objetos de PrintStream**.

La clase **PrintStream** proporciona utilidades para **dar formato a la salida**. Tiene dos métodos **print()** y **println()** que están sobrecargados para los tipos primitivos, objetos, cadenas y arrays de caracteres. La diferencia entre ambos métodos está en que `println` añade un carácter de nueva línea. Además el método `println` además puede llamarse sin argumentos, produciendo una nueva línea.

Otro método de `Printf` es `printf()` que permite **visualizar la salida formateada**.

Algunos de sus constructores de `PrintStream` son los siguientes constructores:

Constructores de PrintStream**PrintStream(File file)**

Crea un flujo de impresión nuevo en el archivo especificado.

PrintStream(OutputStream out)

Crea un flujo de impresión nuevo

PrintStream(OutputStream out, boolean autoFlush)

Crea un flujo de impresión nuevo con vaciado del buffer.

PrintStream(String fileName)

Crea un flujo de impresión nuevo en el archivo especificado.

```

package ejsalidaconsola;
import java.io.PrintStream;
public class EjSalidaConsola {
    public static void main(String[] args) {
        /*El constructor toma como parámetros un objeto de tipo OutputStream del
        cual deriva PrintStream, por tanto, ya tenemos linkada la clase con el dispositivo de
        salida (la consola).
        */
        //Creamos un objeto PrintStream para imprimir en la pantalla
        PrintStream pw = new PrintStream(System.out, true);
        pw.println("Imprime una cadena de texto");
        int i = 15;
        pw.println("Imprime un entero " + i);
        double d = 6.8e-9;
        pw.println("Imprime un double " + d);
        //Utilizamos el objeto ya creado System.out
        //salida formateada
        System.out.printf ("%d, %s, %.2f", 2,"hola ", 8.98987);
        System.out.println();
    }
}

```

Salida:

```

run:
Imprime una cadena de texto
Imprime un entero 15
Imprime un double 6.8E-9
2, hola , 8,99
BUILD SUCCESSFUL (total time: 0 seconds)

```

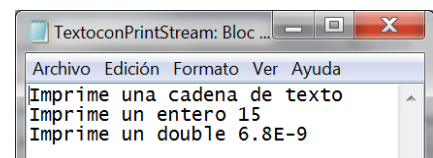
Escritura en un fichero de texto utilizando PrintStream

Si al constructor de PrintStream, en vez de pasarle el objeto System.out, le pasamos un fichero (con File o mediante un String) la salida se graba en el fichero de texto especificado. Tenemos que capturar la excepción **FileNotFoundException**.

```

import java.io.FileNotFoundException;
import java.io.PrintStream;
public class EjEscrituraPrintStream {
    public static void main(String[] args) {
        /*El constructor toma como parámetros un objeto de tipo OutputStream del
        cual deriva PrintStream, por tanto, ya tenemos linkada la clase con el dispositivo de
        salida (la consola).
        */
        PrintStream pw = null;
        try{
            //Creamos un objeto PrintStream para imprimir en la pantalla
            pw = new PrintStream("C:/FicherosJava/TextoconPrintStream.txt");
            pw.println("Imprime una cadena de texto");
            int i = 15;
            pw.println("Imprime un entero " + i);
            double d = 6.8e-9;
            pw.println("Imprime un double " + d);
            //Utilizamos el objeto ya creado System.out
            //salida formateada
            System.out.printf ("%d, %s, %.2f", 2,"hola ", 8.98987);
            System.out.println();
        }
        catch (FileNotFoundException e)
        {
            System.out.printf ("Fichero no encontrado");
            System.out.println();
        }
        finally{
            if (pw!=null)
                pw.close();
        }
    }
}

```



3. Sistema de Ficheros y Directorios en Java

3.1 La clase FILE

La clase **FILE** trabaja **directamente con los archivos y el sistema de archivos**. Se utiliza para obtener o modificar información asociada con un archivo, y para navegar por la jerarquía de subdirectorios.

Un directorio en Java se trata igual que un archivo con una propiedad adicional, una lista de nombres de archivo que se puede examinar utilizando el método LIST.

En el paquete java.io se encuentra la clase File pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

Un objeto File representa un archivo o un directorio y sirve para obtener información (permisos, tamaño,...). También sirve para navegar por la estructura de archivos.

3.1.1 Creación de un objeto File

Los constructores de File permiten inicializar el objeto con el nombre de un archivo y la ruta donde se encuentra. También, inicializar el objeto con otro objeto File como ruta y el nombre del archivo o a través de una URI.

Constructores de la clase FILE	Ejemplos
<pre>public File(String nombreCompleto)</pre> Crea un objeto File con el nombre y ruta del archivo pasado como argumento	<pre>File f1= new File("/carpeta/archivo.txt"); File directorio=new File ("");</pre>
<pre>public File(String ruta, String nombre)</pre> Crea un objeto File en la ruta pasada como primer argumento y con el nombre del archivo como segundo argumento	<pre>File f2 = new File("/", "autoexec.bat");</pre>
<pre>public File(File ruta, String nombre)</pre> Crea un objeto File en la ruta que proporciona un objeto FILE pasada como primer argumento y con el nombre del archivo como segundo argumento.	<pre>File f3 = new File(directorio, "texto.txt");</pre>
<pre>public File(URI uri)</pre> Crea un objeto File a través de un objeto URI.	<pre>File f4= new File (new URI("file:///index.htm"));</pre>

*Nota: Un **Uniform Resource Identifier** o **URI** (en español «identificador uniforme de recurso») es una cadena de caracteres corta que identifica inequívocamente un recurso (servicio, página, documento, dirección de correo electrónico, etc.). Normalmente estos recursos son accesibles en una red o sistema.*

3.1.2 Métodos de la clase File

Modificador y tipo	Método y descripción
boolean	<pre>canExecute ()</pre> Devuelve true si el archivo existe y la aplicación puede ejecutarlo.
boolean	<pre>canRead ()</pre> Devuelve true si el archivo existe y se puede leer.
boolean	<pre>canWrite ()</pre> Devuelve true si el archivo existe y se puede escribir.
int	<pre>compareTo (File ruta)</pre> Compara dos rutas en orden alfabético
boolean	<pre>createNewFile () throws IOException</pre> Crea un nuevo archivo basado en la ruta dada al objeto File si solo si el fichero no existe. Hay que capturar la excepción IOException que ocurriría si hubo error crítico al crear el archivo. Devuelve true si se hizo la creación del archivo vacío y false si ya había otro archivo con ese nombre.
static File	<pre>createTempFile (String prefijo, String sufijo) throws IOException</pre> Crea un fichero vacío en el directorio temporal por defecto usando el prefijo y sufijo para generar el nombre. El prefijo y el sufijo deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo IllegalArgumentException Requiere capturar la excepción IOException que se produce ante cualquier fallo en la creación del archivo

static File	<code>createTempFile(String prefijo, String sufijo, File directorio)</code> throws IOException Crea un fichero vacío en el directorio especificado usando el prefijo y sufijo para generar el nombre.
boolean	<code>delete()</code> Borra el fichero o directorio especificado al crear el objeto File. Devuelve true si puede hacerlo
void	<code>deleteOnExit()</code> Borra el fichero o directorio al finaliza la ejecución del programa.
boolean	<code>equals(Object objeto)</code> Devuelve true si la ruta especificada es igual que el objeto dado.
boolean	<code>exists()</code> Devuelve true si el fichero o directorio especificado existe.
File	<code>getAbsolutePath()</code> Devuelve un objeto File con la ruta absoluta al objeto File creado.
String	<code>getAbsolutePath()</code> Devuelve una cadena con la ruta absoluta al objeto File.
File	<code>getCanonicalFile()</code> Convierte un objeto File a una única forma canónica más adecuada para las comparaciones.
String	<code>getCanonicalPath()</code> Convierte una ruta a una única forma canónica más adecuada para las comparaciones.
long	<code>getFreeSpace()</code> Devuelve el número de bytes libres en la partición.
String	<code>getName()</code> Devuelve el nombre del fichero o directorio especificado.
String	<code>getParent()</code> Devuelve una cadena con el directorio padre o null si no tiene un directorio padre.
File	<code>getParentFile()</code> Devuelve un objeto File con el directorio padre o null si no tiene un directorio padre..
String	<code>getPath()</code> Convierte la ruta especificada al crear el objeto File en una cadena
long	<code>getTotalSpace()</code> Devuelve el espacio total en la partición.
long	<code>getUsableSpace()</code> Devuelve el espacio utilizado por el fichero.
int	<code>hashCode()</code> Computes a hash code for this abstract pathname.
boolean	<code>isAbsolute()</code> Devuelve true si la ruta es absoluta.
boolean	<code>isDirectory()</code> Devuelve true si es un directorio.
boolean	<code>isFile()</code> Devuelve true si es un fichero.
boolean	<code>isHidden()</code> Devuelve true si el fichero es oculto.
long	<code>lastModified()</code> Devuelve la fecha de la última modificación del fichero especificado.
long	<code>length()</code> Devuelve la longitud del fichero especificado.
String[]	<code>list()</code> Devuelve un array de cadenas con los nombres y directorios del directorio especificado.
String[]	<code>list(FilenameFilter filtro)</code> Devuelve un array de cadenas con los nombres y directorios del directorio especificado que satisfacen el filtro.
File[]	<code>listFiles()</code> Lista los archivos que componen el directorio.
File[]	<code>listFiles(FileFilter filtro)</code> Lista los archivos que componen el directorio y que cumplen con el criterio especificado.
File[]	<code>listFiles(FilenameFilter filtro)</code> Lista los archivos que componen el directorio y que cumplen con el criterio especificado.
Static File[]	<code>listRoots()</code> Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
Boolean	<code>mkdir()</code> Crea un directorio en la ruta especificada.
boolean	<code>makedirs()</code> Crea el directorio especificado por el objeto FILE aunque no exista el camino. Todos los directorios anteriores a la carpeta se crearán, si no existe. De este modo no saltará ninguna excepción si es que los directorios no existen.
boolean	<code>renameTo(File destino)</code> Renombra el fichero especificado
boolean	<code>setExecutable(boolean permiso, boolean soloPropietario)</code> Si permiso es true, se establece el permiso de ejecución.
boolean	<code>setExecutable(boolean permiso, boolean soloPropietario)</code> Si permiso es true, se establece el permiso de ejecución. Si soloPropietario es true, el permiso de ejecución solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
boolean	<code>setLastModified(long fecha)</code> Establece el tiempo de la última modificación del archivo o directorio

boolean	<code>setReadable(boolean permiso)</code> Si permiso es true, se permiten operaciones de lectura. Devuelve true si la operación ha tenido éxito.
boolean	<code>setReadable(boolean permiso, boolean soloPropietario)</code> Si permiso es true, se permiten operaciones de lectura. Si soloPropietario es true, el permiso de lectura solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
boolean	<code>setReadOnly()</code> Establece el fichero o directorio de solo lectura.
Boolean	<code>setWritable(boolean permiso)</code> Si permiso es true, se permiten operaciones de lectura y escritura. Devuelve true si la operación ha tenido éxito.
boolean	<code>setWritable(boolean permiso, boolean soloPropietario)</code> Si permiso es true, se permiten operaciones de lectura y escritura. Si soloPropietario es true, el permiso de lectura y escritura solo se aplica al propietario, en otro caso, se aplica a todo el mundo.
Path	<code>toPath()</code> Devuelve un objeto <code>java.nio.file.Path</code> a partir de la ruta especificada
String	<code>toString()</code> Devuelve una cadena a partir de la ruta especificada.
URI	<code>toURI()</code> Devuelve a file: URI que representa la ruta especificada.

Ejemplo:

```
package ejinformacionfichero;
import java.io.*;
import java.util.Date;
import java.text.SimpleDateFormat;
import java.util.Locale;
public class EjInformacionFichero {
    public static void main(String[] args) {
        String ruta= "D:/ACCESO a DATOS";
        // Para formatear la fecha de modificacion con el formato dd de mes de año en español
        SimpleDateFormat formateador =
            new SimpleDateFormat( "dd 'de' MMMM 'de' yyyy", new Locale("es","ES"));
        File f=new File(ruta);
        if(f.exists()){
            System.out.println("\n\tNombre: " + f.getName());
            System.out.println("\tTamaño: " + f.length()+ " bytes");
            System.out.println("\tRuta absoluta: " + f.getAbsolutePath());
            System.out.println("\tPuede leerse: " + f.canRead());
            System.out.println("\tPuede modificar: " + f.canWrite());
            System.out.println("\tEs archivo: " + f.isFile());
            System.out.println("\tEs oculto: " + f.isHidden());
            System.out.println("\tModificado por última vez: " +
                formateador.format(new Date(f.lastModified())));
            System.out.println("\tEs directorio: " + f.isDirectory());
            if(f.isDirectory()){
                System.out.println("\n- ----Contenido del directorio: "+ruta+"- ----");
                /* El método list() devuelve un array con el contenido de un directorio */
                String [] lista=f.list();
                for(int i=0;i<lista.length;i++){
                    System.out.println(ruta+"/"+lista[i]);
                }
            }
            else{
                System.out.println("El "+ruta+" no existe");
            }
        }
    }
}
```

Un ejemplo de salida:

```
Nombre: ACCESO a DATOS
Tamaño: 4096 bytes
Ruta absoluta: D:/ACCESO a DATOS
Puede leerse: true
Puede editarse: true
Es archivo regular: false
Es oculto: false
Modificado por última vez: 17 de abril de 2012
Es directorio: true
Es ejecutable: true

- ----Contenido del directorio: D:/ACCESO a DATOS- ----
D:/ACCESO a DATOS/Ejemplosiniciales
D:/ACCESO a DATOS/NETBEANS 7.1.1
D:/ACCESO a DATOS/MaterialAyuda
D:/ACCESO a DATOS/Unidades didácticas
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Escribir y leer datos de archivos secuenciales binarios: Clases FileOutputStream y FileInputStream

Para acceder a datos almacenados en ficheros sin demasiadas pretensiones de formateo, y con acceso secuencial, puede utilizarse la clase **FileInputStream** que implementa los métodos de la clase **InputStream** sin mayores funcionalidades.

Para almacenar datos en un fichero, puede crearse un stream de salida mediante la clase más simple que implementa los métodos de la clase **OutputStream**: **FileOutputStream**.

FileOutputStream y FileInputStream son clases que manipulan archivos. Son herederas de **Input/OutputStream**, por lo que manejan corrientes de datos en forma de **bytes** binarios.

Excepciones: **FileOutputStream** y **FileInputStream** pueden lanzar la excepción:

- **FileNotFoundException**: Se lanza si el archivo no existe, si es un directorio en lugar de un archivo normal, o por alguna otra razón no se puede abrir para la lectura o para la escritura

Estas clases proporcionan una serie de constructores que permiten indicar el fichero sobre el que se actúa:

Constructores de la clase FileInputStream	Ejemplos
FileInputStream (File fichero) Crea un flujo de entrada FileInputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto <i>File fichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo	<pre>File miFichero = new File("texto.txt"); FileInputStream miFicheroSt = new FileInputStream(miFichero);</pre>
FileInputStream(String nombreFichero) Crea un flujo de entrada FileInputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena <i>nombreFichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo	<pre>FileInputStream miFicheroSt = new FileInputStream("C:\\texto.txt");</pre>
FileInputStream(FileDescriptor DescriptorFichero) Crea un flujo de entrada FileInputStream mediante el descriptor de Fichero.	<pre>FileInputStream Fichero1= new FileInputStream(); FileDescriptor fd = Fichero1.getFD(); FileInputStream Fichero2 = new FileInputStream(fd);</pre>
Constructores de la clase FileOutputStream	Ejemplos
FileOutputStream (File fichero) Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto <i>File fichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo. Si el fichero existe, los datos que contienen se borrarán.	<pre>File miFichero = new File("texto.txt"); FileOutputStream miFicheroSt = new FileOutputStream(miFichero);</pre>
FileOutputStream (File fichero, boolean agregar) Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por el objeto <i>File fichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo. Si el fichero existe y el parámetro <i>agregar</i> es cierto, entonces los bytes se escriben en el final del archivo y no el principio, es decir, no se pierde la información del fichero.	<pre>File miFichero = new File("texto.txt"); FileOutputStream miFicheroSt = new FileOutputStream(miFichero,true);</pre>
FileOutputStream (String nombreFichero) Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena <i>nombreFichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo	<pre>FileOutputStream miFicheroSt = new FileOutputStream("C:\\texto.txt");</pre>
FileOutputStream (String nombreFichero, boolean agregar) Crea un flujo de salida FileOutputStream abriendo una conexión a un archivo en la ruta y nombre indicado por la cadena <i>nombreFichero</i> . Un objeto <i>FileDescriptor</i> se crea para representar a esta conexión de archivo Si el fichero existe y el parámetro <i>agregar</i> es cierto, entonces los bytes se escriben en el final del archivo y no el principio, es decir, no se pierde la información del fichero.	<pre>FileOutputStream miFicheroSt = new FileOutputStream("C:\\texto.txt",true);</pre>
FileOutputStream(FileDescriptor DescriptorFichero) Crea un flujo de salida FileOutputStream mediante el descriptor de Fichero.	<pre>FileOutputStream Fichero1= new FileOutputStream(); FileDescriptor fd = Fichero1.getFD(); FileOutputStream Fichero2 = new FileOutputStream(fd);</pre>

Las clases **FileInputStream** y **FileOutputStream** implementan los métodos **read** y **write** de las clases **InputStream** y **OutputStream**, por lo que no son clases abstractas y podemos crear instancias de las mismas:

Metodos de la clase FileInputStream	
Modificador y tipo	Method and Description
int	available() Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	close() Cierra el fichero y libera los recursos del sistema que esté usando..
protected void	finalize() Asegura que el fichero es cerrado cuando no hay más referencias al él.
FileChannel	getChannel() Retorna el objeto FileChannel asociado con el fichero.
FileDescriptor	getFD() Devuelve el objeto FileDescriptor que representa la conexión actual del fichero en el sistema de ficheros usado por el FileInputStream.
int	read() Lee un byte de datos desde el fichero. Devuelve -1 si no hay ningún byte más que leer.
int	read(byte[] b) No lee un solo byte, sino que lee hasta que b.length bytes guardándolos en el array b que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.
int	read(byte[] b, int off, int len) Lee hasta len bytes del fichero y los deposita en b a partir de off. . Devuelve -1 si no hay ningún byte más que leer.
long	skip(long n) Salta n bytes de datos del fichero.

Metodos de FileOutputStream	
Modificador y tipo	Métodos y descripción
void	close() Cierra el flujo de salida y libera todos los recursos del sistema asociados con esta corriente. Cualquier acceso posterior generaría una IOException
protected void	finalize() Asegura que el método de cierre del flujo de salida del archivo se llame cuando no haya más referencias a este flujo
FileChannel	getChannel() Devuelve el objeto FileChannel único asociado a este flujo de salida del archivo.
FileDescriptor	getFD() Devuelve el descriptor de fichero asociado con el flujo de salida.
void	write(int b) Escribe el byte especificado a en el flujo de salida del archivo.
void	write(byte[] b) Escribe todo el array de bytes en la corriente de salida
void	write(byte[] b, int posinicial, int numbytes) Escribe el array de bytes en el flujo de salida, pero empezando por la posición inicial y sólo la cantidad indicada por numbytes.

Ej: Lectura secuencial byte a byte de un archivo de texto.

```

/*Ejemplo de lectura secuencial byte a byte de un archivo. */
package ejficherossecuencialbytes;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
/**
 * @author María José Galán López
 */
public class EjFicheroSecuencialBytes {
    public static void main(String[] args)
    {
        File f=new File("C:/FicherosJava/ejemplo.txt");
        try {
            // FileInputStream puede lanzar la excepción IOException
            FileInputStream fis=new FileInputStream(f);
            try{
                //método read() puede lanzar la excepción IOException
                int datos;
                datos=fis.read();
                while(datos!=-1)
                {
                    System.out.print((char)datos);
                    datos=fis.read();
                }
                fis.close();
            }
            catch(IOException e)

```

```

        {
            System.out.println("Error en lectura de datos");
        }
    }
    catch (FileNotFoundException e) {
        System.out.println("el fichero "+f.getName()+ " no se encuentra");
    }
}
}

```

Ejemplo: Escritura de forma secuencial en un archivo.

```

package ejescriturasecuencialbytes;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class EjEscrituraSecuencialBytes {
    public static void main(String[] args) throws FileNotFoundException {
        int contLin = 0;
        String lineas[] = {"primera linea",
                           "segunda linea",
                           "tercera linea",
                           "cuarta linea"};

        byte[] s;
        FileOutputStream f=new FileOutputStream("C:/FicherosJava/ejemplo1.txt");

        try {
            for(int i = 0; i < lineas.length;i++){
                // copia la cadena en el array de bytes s;
                s = lineas[i].getBytes();
                f.write(s); // graba el array de bytes
                f.write((byte) '\n');
                contLin++; // cuenta
            }
            System.out.println("Grabadas "+contLin+" lineas (exito)");
            f.close();
        }
        catch (IOException e){
            System.out.println("Problema grabación");
        }
    }
    //Lectura del archivo
    try{
        FileInputStream fis=new FileInputStream("C:/FicherosJava/ejemplo1.txt");
        try{
            int datos;
            datos=fis.read();
            while(datos!=-1)
            {
                System.out.print((char)datos);
                datos=fis.read();
            }
            f.close();
        }
        catch(IOException e)
        {
            System.out.println("Error en lectura de datos");
        }
    }
}
catch (FileNotFoundException e) {
    System.out.println("el fichero no se encuentra");
}
}
}

```

Salida:

```

run:
Grabadas 4 lineas (exito)
primera linea
segunda linea
tercera linea
cuarta linea
BUILD SUCCESSFUL (total time: 0 seconds)

```

5. Las clases orientadas al filtrado del flujo de bytes.

Estas clases proporcionan de manera transparente a estos flujos orientados a byte una funcionalidad de un nivel superior dado que las clases orientadas a flujo de bytes quedan algo limitadas para "trabajar" sobre los datos leídos o sobre los datos que vamos a escribir.

Los stream filtro son una abstracción de las secuencias de bytes para hacer procesos de datos a más alto nivel; con esta abstracción ya no tratamos los items como secuencias o «chorros» de bytes, sino de forma elaborada con más funcionalidad. Así, a nivel lógico, se pueden tratar los datos dentro de un buffer, escribir o leer datos de tipo int, long, double directamente y no mediante secuencias de bytes. Los objetos stream filtro leen de un flujo que previamente ha tenido que ser escrito por otro objeto stream filtro de salida.

Las clases más representativas de este tipo son **FilterInputStream y FilterOutputStream**:

<pre> java.io Class FilterInputStream java.lang.Object java.io.InputStream java.io.FilterInputStream All Implemented Interfaces: Closeable, AutoCloseable Direct Known Subclasses: BufferedInputStream, CheckedInputStream, CipherInputStream, DataInputStream, DeflaterInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream </pre>	<pre> java.io Class FilterOutputStream java.lang.Object java.io.OutputStream java.io.FilterOutputStream All Implemented Interfaces: Closeable, Flushable, AutoCloseable Direct Known Subclasses: BufferedOutputStream, CheckedOutputStream, CipherOutputStream, DataOutputStream, DeflaterOutputStream, DigestOutputStream, InflaterOutputStream, PrintStream </pre>
---	---

Los filtros devuelven la información que a su vez han leído de su InputStream o la escriben en su OutputStream asociado, previa realización de algún tipo de transformación en la misma.

- De esta manera cada filtro añade una funcionalidad adicional al InputStream o OutputStream básico.
- Se pueden encadenar varios filtros para obtener varias funcionalidades combinadas

5.1 Acceso a datos primitivos: Clases DataInputStream y DataOutputStream

Aunque leer y escribir bytes es útil, a menudo es necesario transmitir datos de tipos primitivos dentro de un flujo. Las clases DataInputStream y DataOutputStream proporcionan métodos para la lectura y escritura de tipos primitivos (int, float, double, etc..) de un modo independiente de la máquina.

```
public class DataOutputStream
extends FilterOutputStream
implements DataOutput
```

```
public class DataInputStream
extends FilterInputStream
implements DataInput
```

Constructor de DataOutputStream <pre>DataOutputStream(OutputStream sal)</pre> <p>Se crea un objeto de la clase DataOutputStream vinculándolo a un objeto OutputStream para escribir en un archivo. Permite escribir un flujo de salida, datos de cualquier tipo primitivo (int, float, double, etc..)</p>	Ejemplo: <pre>FileOutputStream fileSal=new FileOutputStream("pedido.txt"); DataOutputStream Salida=new DataOutputStream(fileSal);</pre>
Constructor de DataInputStream <pre>DataInputStream(InputStream in)</pre> <p>Se crea un objeto de la clase DataInputStream vinculándolo a un objeto InputStream para leer desde un archivo. Permite leer un flujo de entrada, datos de cualquier tipo primitivo. Solo es posible leer datos, escritos por DataOutputStream</p>	Ejemplo: <pre>FileInputStream fileEnt=new FileInputStream("pedido.txt"); DataInputStream entrada=new DataInputStream(fileEnt);</pre>

A continuación se detallan los métodos de estas dos clases:

Modificador y tipo de DataOutputStream	Métodos y descripción
void	flush() throws IOException Vacía el buffer de salida. Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
int	size() Devuelve el número de bytes escritos en esta corriente de datos de salida hasta el momento.
void	write(byte[] b, int off, int len) throws IOException Escribe len bytes en la corriente de salida desde el array b comenzando en la posición off.
void	write(int b) throws IOException Escribe el primer byte de menor peso a corriente de salida. Los otros 3 restantes bytes se ignoran..
void	writeBoolean(boolean v) throws IOException Escribe un char en el flujo de salida ocupando 1 byte .
void	writeByte(int v) throws IOException Escribe un byte en el flujo de salida ocupando 1 byte.
void	writeBytes(String s) throws IOException Escribe una cadena en el flujo de salida como una secuencia de bytes.
void	writeChar(int v) throws IOException Escribe un char en el flujo de salida ocupando 2 bytes.
void	writeChars(String s) throws IOException Escribe una cadena en el flujo de salida como una secuencia de caracteres.
void	writeDouble(double v) throws IOException Escribe un valor numérico double en el flujo de salida ocupando 8bytes.
void	writeFloat(float v) throws IOException Escribe un valor numérico float en el flujo de salida ocupando 4bytes.
void	writeInt(int v) throws IOException Escribe un valor numérico entero en el flujo de salida ocupando 4bytes.
void	writeLong(long v) throws IOException Escribe un valor entero long en el flujo de salida ocupando 8bytes.
void	writeShort(int v) throws IOException Escribe un valor entero short en el flujo de salida ocupando 2bytes.
void	writeUTF(String str) throws IOException Escribe una cadena UNICODE utilizando codificación UTF-8 . en el flujo de salida.

Modificador y tipo de DataInputStream	Métodos y descripción
int	read(byte[] b) throws IOException Lee b.length bytes del flujo de entrada y los guarda en el array pasado como parámetro. Devuelve el número total de bytes leídos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada.
int	read(byte[] b, int off, int len) throws IOException Lee len bytes del flujo de entrada y los guarda en el array a partir del desplazamiento off pasado como parámetro. Devuelve el número total de bytes leídos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada
boolean	readBoolean() throws IOException Lee un byte del flujo de entrada y devuelve verdadero si ese byte es distinto de cero, falso si ese byte es cero. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
byte	readByte() Lee un byte del flujo de entrada. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
char	readChar() throws IOException Lee dos bytes del flujo de entrada y devuelve un valor char. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
double	readDouble() throws IOException Lee ocho bytes del flujo de entrada y devuelve un valor numérico double. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
float	readFloat() throws IOException Lee cuatro bytes de entrada y devuelve un valor numérico float Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
void	readFully(byte[] b) throws IOException Lee b.length bytes del flujo de entrada y los deposita en el array b. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
void	readFully(byte[] b, int off, int len) throws IOException Lee b.length bytes del flujo de entrada y los deposita en el array b a partir del desplazamiento off. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
int	readInt() throws IOException Lee cuatro bytes de entrada y devuelve un valor numérico int. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
String	readLine() throws IOException Deprecated. <i>Este método no convierte correctamente los bytes a caracteres. A partir de JDK 1.1, la mejor forma de leer las líneas de texto se realiza mediante el método <code>BufferedReader.readLine()</code>. Los programas que utilizan la clase <code>DataInputStream</code> para leer las líneas se pueden sustituir el código de la forma:</i> <i>DataInputStream d = new DataInputStream(en);</i> <i>Por:</i> <i>BufferedReader d = new BufferedReader(new InputStreamReader(en))</i>

long	readLong() throws IOException Lee ocho bytes de entrada y devuelve un valor numérico long. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
short	readShort() throws IOException Lee dos bytes de entrada y devuelve un valor numérico short. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
int	readUnsignedByte() throws IOException Lee un byte del flujo de entrada y devuelve un valor int en el rango de valores del 0 al 255. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
int	readUnsignedShort() throws IOException Lee dos bytes del flujo de entrada y devuelve un valor int en el rango de valores del 0 al 65535. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
String	readUTF() throws IOException Lee una cadena Unicode en formato UTF-8 formato del flujo de entrada. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
static String	readUTF(DataInput in) throws IOException Lee una cadena Unicode en formato UTF-8 formato del flujo de entrada. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes
int	skipBytes(int n) throws IOException Hace un intento de saltarse n bytes de datos del flujo de entrada, descarta los bytes omitidos. Sin embargo, puede saltar sobre un número menor de bytes, posiblemente cero. Este método no produce una EOFException.

Ej:

```

package ejficheroSecuencialDatosPrimitivos;
import java.io.*;
public class EjFicheroSecuencialDatosPrimitivos {
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        double[] precios={1.35, 4.0, 8.90, 6.2, 8.73};
        int[] unidades={5, 7, 12, 8, 30};
        String[] descripciones={"paquetes de papel", "lápices", "bolígrafos",
                                "carteras", "mesas"};
        DataOutputStream salida=new DataOutputStream
            (new FileOutputStream("C:/FicherosJava/pedido.dat"));
        for (int i=0; i<precios.length; i++) {
            salida.writeBytes(descripciones[i]);
            salida.writeChar('\n');
            salida.writeInt(unidades[i]);
            salida.writeChar('\t');
            salida.writeDouble(precios[i]); }
        double precio;
        int unidad;
        char car;
        String descripcion;
        double total=0.0;
        DataInputStream entrada = new DataInputStream(new
        FileInputStream("C:/FicherosJava/pedido.dat"));
        try {
            while (true)
            { //para leer la cadena de caracteres.
              //se podría hacer con readline pero está deprecated.
              descripcion="";
              car=(char)entrada.readUnsignedByte(); //para que lea las vocales
              acentuadas (lee de 0 a 255, sino leeria de 0 a 128)
              while (car!='\n'){
                  descripcion+=car;
                  car=(char)entrada.readUnsignedByte(); }
              unidad=entrada.readInt();
              entrada.readChar(); //lee el carácter tabulador
              precio=entrada.readDouble();
              System.out.println("has pedido "+unidad+" "+descripcion+" a
              "+precio+" Euros.");
              total=total+unidad*precio;
            }
        }
        catch (EOFException e) {
            // Es para controlar que se ha llegado al final del archivo }
            System.out.printf("%s %.2f %s\n","por un TOTAL de ",total," pts.");
            entrada.close();
        }
    }
}

```

5.2 Flujos orientados a byte con buffer: Clases `BufferedInputStream` y `BufferedOutputStream`

Estas clases asignan un buffer de memoria a los flujos de byte de I/O. Este buffer le permite a Java realizar operaciones de I/O sobre más de un byte a la misma vez con lo que estas clases incrementan y optimizan las prestaciones a la hora de trabajar con estos datos de esta forma, ya que se reduce el número de veces que el sistema realmente accede al dispositivo. La diferencia es clara, no es lo mismo leer o escribir byte a byte que leer o enviar un flujo de N bytes (tamaño del buffer) de una tacada desde o al dispositivo.

Por tanto, esta clase nos permite "envolver" cualquier `InputStream` u `OutputStream` en un stream con buffer para optimizar el uso de la memoria.

Constructor de <code>BufferedInputStream</code>	
<code>BufferedInputStream(InputStream in)</code>	Crea un objeto <code>BufferedInputStream</code> con el tamaño de buffer por defecto
<code>BufferedInputStream(InputStream in, int size)</code>	Crea un objeto <code>BufferedInputStream</code> con el tamaño de buffer especificado en el argumento <code>size</code> . Lanza una <code>IllegalArgumentException</code> si <code>size <= 0</code>
Constructor de <code>BufferedOutputStream</code>	
<code>BufferedOutputStream(OutputStream out)</code>	Crea un objeto <code>BufferedOutputStream</code> con el tamaño de buffer por defecto
<code>BufferedOutputStream(OutputStream out, int size)</code>	Crea un objeto <code>BufferedOutputStream</code> con el tamaño de buffer especificado en el argumento <code>size</code> . Lanza una <code>IllegalArgumentException</code> si <code>size <= 0</code> .

Metodos de <code>BufferedInputStream</code>	
int	<code>available()</code> throws <code>IOException</code> Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	<code>close()</code> throws <code>IOException</code> Cierra la corriente de entrada y libera los recursos del sistema que esté usando
void	<code>mark(int readlimit)</code> Marca la posición actual de la corriente de entrada.
boolean	<code>markSupported()</code> Prueba si esta corriente de entrada soporta los métodos <code>mark</code> y <code>reset</code> .
int	<code>read()</code> throws <code>IOException</code> Lee el siguiente byte de la corriente de entrada.
int	<code>read(byte[] b, int off, int len)</code> throws <code>IOException</code> Lee <code>b.length</code> bytes del flujo de entrada y los guarda en el array pasado como parámetro. Devuelve el número total de bytes leídos o -1 si no hay más datos debido a que se alcanza el final del flujo de entrada.
void	<code>reset()</code> throws <code>IOException</code> Coloca la corriente de entrada en la posición que tenía la última vez que se invocó <code>mark</code> .
long	<code>skip(long n)</code> throws <code>IOException</code> Salta y descarta los siguientes <code>n</code> bytes de la corriente de entrada.
Métodos heredados de la clase <code>java.io.FilterInputStream</code> : <code>read</code>	

Metodos de <code>BufferedOutputStream</code>	
void	<code>flush()</code> throws <code>IOException</code> Vacía el buffer de salida. Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
void	<code>write(byte[] b, int off, int len)</code> throws <code>IOException</code> Escribe <code>len</code> bytes en la corriente de salida desde el array <code>b</code> comenzando en la posición <code>off</code> .
void	<code>write(int b)</code> throws <code>IOException</code> Escribe el primer byte de menor peso a corriente de salida. Los otros 3 restantes bytes se ignoran..
Metodos heredados de la clase <code>java.io.FilterOutputStream</code> : <code>close</code> y <code>write</code>	

Ej: Leer un fichero de texto utilizando un flujo de bytes con buffer.

```
public class EjFicheroSecuencialByteconBuffer {
    public static void main(String[] args) throws IOException {
        File f=new File("C:/FicherosJava/ejemplo.txt");
        FileInputStream fis=null;
        BufferedInputStream entrada=null;
        try {
            fis=new FileInputStream(f);
            entrada=new BufferedInputStream(fis);
            try{
                int datos;
                datos=entrada.read();
                while(datos!=-1)
                { System.out.print((char)datos);
                  datos=entrada.read();}
            }
        }
    }
}
```

```

        catch(IOException e)
        {System.out.println("Error en lectura de datos");}
    }
}
catch (FileNotFoundException e) {
    System.out.println("el fichero "+f.getName()+ " no se encuentra");
}
finally{
    if (fis!=null) fis.close();
    if (entrada!=null)entrada.close();}
}
}

```

5.3 Combinación de clases sobre los flujos de entrada y salida

A continuación veremos un ejemplo que combinan las clases que tratan directamente sobre los flujos de entrada o de salida con las clases que las envuelven.

Ej: leer un archivo de números reales con buffer.

```

public class EjFicheroDatosRealesconbuffer {
    public static void main(String[] args) throws IOException {
        File f=new File("C:/FicherosJava/FicheroDoublebuffer.bin");
        Random r;
        FileOutputStream fis=null;
        DataOutputStream salida=null;
        BufferedOutputStream buffer=null;
        try{
            fis=new FileOutputStream(f);
            buffer=new BufferedOutputStream(fis);
            salida=new DataOutputStream(buffer);
            for (int i=0;i<100;i++){
                r=new Random();//Se repite 233 veces
                salida.writeDouble(r.nextDouble());//Nº aleatorio
            }
        }
        catch(FileNotFoundException e){
            System.out.println("No se encontro el archivo");
        }
        catch(IOException e){
            System.out.println("Error al escribir");
        }
        finally{
            if (salida!=null) salida.close();
            if (buffer!=null) buffer.close();
            if (fis!=null) fis.close();
        }
        System.out.println( "lectura del archivo binario de numeros reales con
            buffer");
        FileInputStream fie=null;
        BufferedInputStream buffer2=null;
        DataInputStream entrada=null;
        try{
            fie=new FileInputStream(f);
            buffer2=new BufferedInputStream(fie);
            entrada=new DataInputStream(buffer2);
            try{
                double datos; int i=1;
                datos=entrada.readDouble();
                while (true){
                    System.out.printf( "%6.2f%c",datos,',');
                    datos=entrada.readDouble();
                    i++;
                }
            }catch(EOFException e){ //para detectar final de fichero
                System.out.println();
                System.out.println("final del archivo");
            }
        }
        catch(FileNotFoundException e){
            System.out.println("No se encontro el archivo");
        }
        catch(IOException e){
            System.out.println("Error al leer");
        }
        finally{
            if (entrada!=null) entrada.close();
            if (buffer2!=null) buffer2.close();
            if (fie!=null) fie.close();
        }
    }
}

```

6. Escribir y leer datos en archivos secuenciales de texto.

6.1 Clases FileReader y FileWriter

Existen dos clases que **manejan caracteres** en lugar de bytes en ficheros (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader**. Estas clases:

- Convierten cada carácter de la codificación del sistema operativo nativo a código Unicode.
- La mayoría de las clases de flujos orientados a byte tienen su correspondiente clase de flujo orientado a carácter:
- **FileReader**: es el equivalente a `FileInputStream`. Es una clase abstracta que define el modelo de Java de entrada de caracteres.
- **FileWriter** es el equivalente a `FileOutputStream`. Es una clase abstracta que define el modelo de salida de caracteres.



- Como ocurría con la entrada estándar, se puede convertir un objeto `FileInputStream` o `FileOutputStream` a forma de `Reader` o `Writer` mediante las clases **InputStreamReader** y **OutputStreamWriter**. Y esto es más lógico cuando manejamos archivos de texto.

La construcción de objetos del tipo `FileReader` se hace con un parámetro que puede ser un objeto `File` o un `String` o el descriptor del fichero que representarán a un determinado archivo.

Constructor de FileReader
<code>FileReader(File file)</code> throws <code>IOException</code> Lee caracteres del fichero pasado como argumento como un objeto <code>File</code> .
<code>FileReader(FileDescriptor fd)</code> Lee caracteres del fichero pasado como argumento el descriptor del fichero.
<code>FileReader(String fileName)</code> throws <code>IOException</code> Lee caracteres del fichero pasado como argumento como una cadena.

La construcción de objetos `FileWriter` se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer `true`, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Constructor de FileWriter
<code>FileWriter(File file)</code> throws <code>IOException</code> Escribe caracteres en el fichero pasado como argumento como un objeto <code>File</code> .
<code>FileWriter(File file, boolean append)</code> throws <code>IOException</code> Escribe caracteres en el fichero pasado como argumento como un objeto <code>File</code> . Si el argumento <code>append</code> es <code>true</code> , abre el fichero para añadir datos, en caso contrario, se grabaría desde el principio borrando el contenido si existe el fichero
<code>FileWriter(FileDescriptor fd)</code> Escribe caracteres en el fichero pasado como argumento el descriptor del fichero.
<code>FileWriter(String fileName)</code> throws <code>IOException</code> Escribe caracteres en el fichero pasado como argumento una cadena.
<code>FileWriter(String fileName, boolean append)</code> throws <code>IOException</code> Escribe caracteres en el fichero pasado como argumento una cadena. Si el argumento <code>append</code> es <code>true</code> , abre el fichero para añadir datos, en caso contrario, se grabaría desde el principio borrando el contenido si existe el fichero.

- Los **métodos de FileReader** son los heredados de las clases *Object*, *Reader* e *InputStreamReader*:

Method Summary
Methods inherited from class java.io.InputStreamReader close, getEncoding, read, read, ready
Methods inherited from class java.io.Reader mark, markSupported, read, read, reset, skip
Methods inherited from class java.lang.Object clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

- Los **métodos de FileWriter** son los heredados de las clases *Object*, *Writer* y *OutputStreamWriter*:

Method Summary
Methods inherited from class java.io.OutputStreamWriter close, flush, getEncoding, write, write, write
Methods inherited from class java.io.Writer append, append, append, write, write
Methods inherited from class java.lang.Object clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

6.2 Codificación de caracteres

En este apartado vamos a tratar los errores que tenemos habitualmente con la codificación de caracteres en aplicaciones en las que se ven implicados varios sistemas que intercambian o almacenan información.

Codificación de caracteres.

La codificación de caracteres es el método que **permite convertir un caracter del language natural**, el de los humanos, en **un símbolo de otro sistema de representación, aplicando una serie de normas** o reglas de codificación.

El ejemplo más gráfico suele ser el del código morse, cuyas reglas permiten convertir letras y números en señales (rayas y puntos) emitidas de forma intermitente.

En informática, las normas de codificación permiten que dos sistemas intercambien información usando el mismo código numérico para cada caracter.

Las normas más conocidas de codificación son las siguientes:

- ASCII:** basado en el alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales. **Utiliza 7 bits** para representar los caracteres, aunque inicialmente empleaba un bit adicional (bit de paridad) que se usaba para detectar errores en la transmisión. Incluye, básicamente, letras mayúsculas y minúsculas del inglés, dígitos, signos de puntuación y caracteres de control, dejando fuera los caracteres específicos de los idiomas distintos del inglés, como por ejemplo, las vocales acentuadas o la letra ñ.
- ISO-8859-1 (Latin-1):** es una **extensión del código ascii que utiliza 8 bits** para proporcionar caracteres adicionales usados en idiomas distintos al inglés, como el español. Existen 15 variantes y cada una cubre las necesidades de un alfabeto diferente: latino, europa del este, hebreo cirílico,... la norma **ISO-8859-15, es el Latin-1, con el caracter del euro.**
- cp1252** (codepage 1252): Windows usa sus propias variantes de los estándares ISO. La cp1252 es compatible con ISO-8859-1, menos en los 32 primeros caracteres de control, que han usado para incluir, por ejemplo, el caracter del euro.
- UTF-8:** es el formato de transformación **Unicode, de 8 bits de longitud variable**. Unicode es un estándar industrial cuyo objetivo es proporcionar el medio por el cual un texto en cualquier forma e idioma pueda ser codificado para el uso informático. Cubre la mayor parte de las escrituras usadas actualmente.

En la enumeración hemos ido de menos a más, no solo en el tiempo, por el momento de aparición de la norma, sino también por los caracteres que soporta cada una, UTF-8 es la más ambiciosa.

Visto así, la recomendación debería ser el uso de UTF-8 puesto que, escriba en la lengua que escriba, sus caracteres van a ser codificables. Pero, si sólo escribo en castellano, podría limitarme a usar ISO-8859-1, o ISO-8859-15 si necesito el caracter del euro, sin ningún problema.

Importante:

La misma norma de codificación que se use para escribir se debe utilizar para la lectura.

Esto tiene toda la lógica del mundo, puesto que si escribimos un fichero con ISO-8859-1 no debemos esperar que un sistema que lee en UTF-8 lo entienda sin más (aunque realmente entienda gran parte).

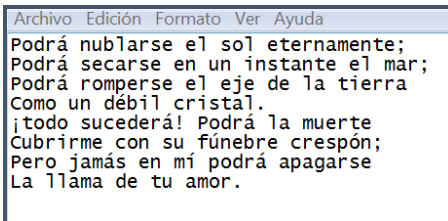
¿Por qué aparecen caracteres "raros"?:

Los caracteres "raros" aparecen por una conversión incorrecta entre dos codificaciones distintas. Se suelen producir porque se utiliza la codificación por defecto del sistema o programa y esta no coincide con la original o, directamente, por desconocimiento de la norma de codificación de la fuente de lectura.

Así, por ejemplo, podemos encontrarnos con los siguientes caracteres "raros" escribiendo la misma palabra:

- **España → Españ±a: si escribimos en UTF-8 y leemos en ISO-8859-1.** La letra ñe se codifica en UTF-8 con dos bytes que en ISO-8859-1 representan la A mayúscula con tilde (Ã) y el símbolo más-menos (±).
- **España → Espãa: si escribimos en ISO-8859-1 y leemos en UTF-8.** La codificación de la ñe en ISO-8859-1 es inválida en UTF-8 y se sustituye por un caracter de sustitución, que puede ser una interrogación, un espacio en blanco... depende de la implementación.

Ejemplo: Tenemos un fichero de texto creado con el bloc de notas de Windows con codificación ASCII

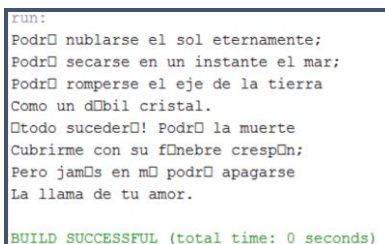


Si intentamos leerlo con FileWiter, los acentos nos saldrán caracteres raros por que los métodos de esta clase leen caracteres Unicode.

Ejemplo:

```
public class EJCaracterAcentuadas {
    public static void main(String[] args) throws IOException {
        FileReader entrada=null;
        try{
            int cad;
            entrada= new FileReader("C:/FicherosJava/Copia.txt");
            while ((cad = entrada.read()) != -1)
            {
                System.out.print((char)cad);
            }
        } catch(FileNotFoundException e){
            System.out.println("error al abrir el fichero");
        } catch(IOException e){
            System.out.printf("error de entrada/salida");
        } finally{
            if (entrada!=null)
                entrada.close();
        }
    }
}
```

Salida:



Para solucionarlo, podemos abrir el fichero para que se use, en este caso, la codificación **ISO-8859-15**

Las clases **InputStreamReader** e **OutputStreamReader**:

- Representan una conexión entre un stream de bytes y un stream de caracteres.
- Podemos leer bytes convertirlos a caracteres atendiendo a una codificación concreta (ISO Latin 1, UTF8,...).

6.3 Las clases **OutputStreamReader** e **InputStreamReader**.

La librería de io de Java proporciona estas clases: **InputStreamReader**, que sirve de puente entre un **InputStream** y un **Reader**, y **OutputStreamReader**, que hace lo mismo entre un **OutputStream** y un **Writer**.

Constructor de InputStreamReader	
<code>InputStreamReader(InputStream in)</code>	Crea un objeto <code>InputStreamReader</code> que usa la codificación de caracteres por defecto
<code>InputStreamReader(InputStream in, Charset cs)</code>	Crea un objeto <code>InputStreamReader</code> que usa la codificación pasada como argumento
<code>InputStreamReader(InputStream in, CharsetDecoder dec)</code>	Crea un objeto <code>InputStreamReader</code> que usa la codificación pasada como argumento
<code>InputStreamReader(InputStream in, String charsetName)</code>	Crea un objeto <code>InputStreamReader</code> que usa la codificación pasada como argumento
Constructor de OutputStreamReader	
<code>OutputStreamWriter(OutputStream out)</code>	Crea un objeto <code>OutputStreamWriter</code> que usa la codificación de caracteres por defecto.
<code>OutputStreamWriter(OutputStream out, Charset cs)</code>	Crea un objeto <code>OutputStreamWriter</code> que usa la codificación pasada como argumento
<code>OutputStreamWriter(OutputStream out, CharsetEncoder enc)</code>	Crea un objeto <code>OutputStreamWriter</code> que usa la codificación pasada como argumento
<code>OutputStreamWriter(OutputStream out, String charsetName)</code>	Crea un objeto <code>OutputStreamWriter</code> que usa la codificación pasada como argumento de cadena.

Metodos de InputStreamReader	
void	<code>close()</code> Cierra el fichero y libera sus recursos asociados.
String	<code>getEncoding()</code> Devuelve el nombre de la cdificación usada por el flujo.
int	<code>read()</code> Lee un caracter
int	<code>read(char[] cbuf, int off, int len)</code> Lee len caracteres de la corriente de entrada y los deposita en el vector de caracteres cbuf a partir de off
boolean	<code>ready()</code> Indica si el flujo de entrada está listo para ser leído
Metodos de OutputStreamReader	
void	<code>close()</code> cierra el flujo de salida y libera sus recursos asociados.
void	<code>flush()</code> Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
String	<code>getEncoding()</code> Devuelve el nombre de la cdificación usada por el flujo.
void	<code>write(char[] cbuf, int off, int len)</code> Escribe len caractees a partir del desplazamineto marcado por off.
void	<code>write(int c)</code> Escribe un caracter
void	<code>write(String str, int off, int len)</code> Escribe una porción de la cadena empezando en off y de longitud len.

Ej: de lectura de un fichero con codificación ASCII

```
public static void main(String[] args) throws IOException {
    InputStreamReader ent;
    try{
        ent=new InputStreamReader(new FileInputStream
                                ("C:/FicherosJava/Copia.txt"), "ISO-8859-15");
        System.out.println(ent.getEncoding());
        int cad;
        while ((cad = ent.read()) != -1)
            System.out.println((char)cad)
    }
}
```

```

    }
    catch (FileNotFoundException e){
        System.out.println("error al abrir el fichero"); }
    catch (IOException e){
        System.out.printf("error de entrada/salida"); }
    finally{
        if (ent!=null)
            ent.close(); }
}

```

Salida:

```

1.1.1.1
ISO8859_15
Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
Como un débil cristal.
¡todo sucederá! Podrá la muerte
Cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
La llama de tu amor.

BUILD SUCCESSFUL (total time: 0 seconds)

```

6.4 Buffer para el flujo de caracteres: Clases `BufferedReader` y `BufferedWriter`

A la hora de optimizar los flujos de caracteres, existen las clases `BufferedReader` y `BufferedWriter` que crean un buffer para agilizar las operaciones de lectura y escritura en flujos de caracteres.

Son análogas a las de flujo de bytes: `BufferedInputStream` y `BufferedOutputStream`.

- La clase **`BufferedReader`** recibe un flujo de caracteres e implementa un buffer para poder leer líneas de texto.

Define el método **`readLine`** para leer una línea de texto. Esta clase se utiliza si sabemos que el archivo es de texto y está escrito en líneas separadas por retornos de carro.

Constructor de <code>BufferedReader</code>
<code>BufferedReader (Reader in)</code> Crea un buffer para el flujo de caracteres de entrada utilizando el tamaño por defecto.
<code>BufferedReader (Reader in, int sz)</code> Crea un buffer para el flujo de caracteres de entrada utilizando el tamaño indicado en el parámetro sz. Lanza un <code>IllegalArgumentException</code> si sz es <= 0

Para el manejo de archivos hay dos formas para construir un objeto de tipo `BufferedReader`:

- Una es utilizar un objeto `InputStreamReader` creado sobre un objeto de tipo `FileInputStream`:

```

BufferedReader buffer = new BufferedReader (new InputStreamReader (
    new FileInputStream ("archivo.dat")));

```

- La otra forma es aceptar el tamaño del buffer y la codificación predefinidos, lo cuál muchas veces es lo más conveniente, y para estos casos se puede usar la clase `FileReader` que como argumento en el constructor se le pasa el nombre del archivo.

```

BufferedReader fd_in = new BufferedReader ( new FileReader ("archivo.dat"));

```

- La clase **`BufferedWriter`** escribe texto a un flujo de salida que acepte caracteres proporcionando un buffer para la escritura eficiente de caracteres, arreglos y strings.

Define el método **`write`** para escribir una línea de texto y el método **`newLine`** para escribir un salto de línea de acuerdo al sistema operativo. Esta clase se utiliza si sabemos que el archivo es de texto y está escrito en líneas separadas por retornos de carro.

Constructor de BufferedWriter**BufferedWriter** (**Writer** out)

Crea un buffer para el flujo de caracteres de salida utilizando el tamaño por defecto.

BufferedWriter (**Writer** out, int sz)Crea un buffer para el flujo de caracteres de salida utilizando el tamaño indicado en el parámetro sz. Lanza un **IllegalArgumentException** si sz es <= 0

Para el manejo de archivos hay dos formas para construir un objeto de tipo **BufferedWriter**:

- Una es utilizar un objeto **OutputStreamWriter** creado sobre un objeto de tipo **FileOutputStream**:

```
BufferedWriter fd_out = new BufferedWriter (new OutputStreamWriter
                                         (new FileOutputStream ("archivo.dat")));
```

La ventaja de hacerlo así es que se puede manejar el tamaño del buffer e incluso cambiar la codificación de los caracteres.

- La otra forma es aceptar el tamaño del buffer y la codificación predefinidos, lo cuál muchas veces es lo más conveniente, y para estos casos se puede usar la clase **FileWriter** que como argumento en el constructor se le pasa el nombre del archivo.

```
BufferedWriter fd_out = new BufferedWriter (new FileWriter ("archivo.dat"));
```

Ej de copiar un fichero de Texto ASCII en otro fichero utilizando buffers para mejorar las operaciones de lectura y escritura:

```
public class EjficheroTextoBuffer{
    public static void main(String[] args) throws IOException {
        InputStreamReader entrada=null;
        OutputStreamWriter salida=null;
        BufferedReader bufferent=null;
        BufferedWriter buffersal=null;
        try{
            salida= new OutputStreamWriter(new FileOutputStream(
                "C:/FicherosJava/Destino.txt"), "ISO-8859-15");
            buffersal=new BufferedWriter(salida);
            String cad;
            entrada= new InputStreamReader
                (new FileInputStream( "C:/FicherosJava/Copia.txt"), "ISO-8859-15");
            bufferent=new BufferedReader(entrada);
            while ((cad = bufferent.readLine()) != null)
            {
                System.out.println(cad);
                salida.write(cad);
            }
        }catch(FileNotFoundException e){
            System.out.println("error al abrir el fichero");
        }
        catch(IOException e){
            System.out.printf("error de entrada/salida");
        }
        finally{
            if (entrada!=null) entrada.close();
            if (salida!=null) salida.close();
            if (buffersal!=null) buffersal.close();
            if (bufferent!=null) bufferent.close();
        }
    }
}
```

7. Escritura formateada: la clase **PrintWriter**

Java también proporciona la clase **PrintWriter** para escribir en un fichero de texto utilizando flujos de caracteres. Su análogo en flujo de bytes es **PrintStream** y uso es muy parecido

El constructor de la clase **PrintWriter** recibe un objeto de **File** y puede lanzar la excepción **FileNotFoundException** si se produce algún problema para su apertura.

Constructor

```

PrintWriter(File file)
Crea un flujo de caracteres en el fichero especificado File.

PrintWriter(OutputStream out)
Crea un flujo de caracteres en el fichero desde un OutputStream existente

PrintWriter(OutputStream out, boolean autoFlush)
Crea un flujo de caracteres en el fichero desde un OutputStream existente con vaciado de
buffer.

PrintWriter(String fileName)
Crea un flujo de caracteres en el fichero especificado en el argumento String

PrintWriter(Writer out)
Crea un flujo de caracteres en el fichero desde un Writer existente

PrintWriter(Writer out, boolean autoFlush)
Crea un flujo de caracteres en el fichero desde un Writer existente

```

A igual que `PrintStream`, tiene los dos métodos para escritura:

- **print(Tipo t);** Escribe un valor `t` de tipo `Tipo`, donde `Tipo` puede ser `char`, `boolean`, `char[]`, `double`, `int`, `String`, `long`, `float` u `Object`
- **println(Tipo p)** Escribe un objeto de tipo `boolean`, `char`, `char[]`, `String`, `double`, `float`, `int`, `long` u `Object` y después salta de línea.

```

import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class EjEscrituraPrintWriter {
    public static void main(String[] args) {
        /*El constructor toma como parámetros un objeto de tipo OutputStream del
        cual deriva PrintStream, por tanto, ya tenemos linkada la clase con el dispositivo de
        salida (la consola).
        */
        PrintWriter pw = null;
        try{
            //Creamos un objeto PrintStream para imprimir en la pantalla
            pw = new PrintWriter("C:/FicherosJava/TextoconPrintWriter.txt");
            pw.println("Imprime una cadena de texto");
            int i = 15;
            pw.println("Imprime un entero " + i);
            double d = 6.8e-9;
            pw.println("Imprime un double " + d);
            //Utilizamos el objeto ya creado System.out
            //salida formateada
            System.out.printf ("%d, %s, %.2f", 2,"hola ", 8.98987);
            System.out.println();
        }
        catch (FileNotFoundException e)
        {
            System.out.printf ("Fichero no encontrado");
            System.out.println();
        }
        finally{
            if (pw!=null)
                pw.close();
        }
    }
}

```

8. La clase StreamTokenizer

`StreamTokenizer` te sirve para leer de un archivo o un flujo de entrada y convertir ese flujo en tokens tomando en cuenta caracteres válidos tales como: **espacios, comas, punto, punto y coma etc...**

Cuando se le envía un mensaje de **nextToken()** a un `StreamTokenizer` se queda esperando a que lleguen bytes por el flujo que se le ha asociado en su creación. Conforme va llegando ese flujo se espera a que se pueda completar una palabra o un número. En cuanto se tiene una de las dos, la ejecución continúa con la siguiente instrucción a `nextToken()`.

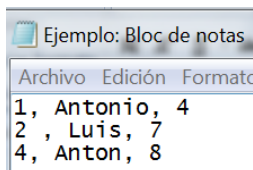
`StreamTokenizer` permite seleccionar el carácter que se utiliza como separador de tokens, como fin de línea e incluso identificar comentarios que no se asocian a ningún token. Cuando el stream proviene de fichero la condición de EOF (end of file) se produce de manera natural, sin embargo cuando el stream proviene del teclado es necesario forzar EOF mediante la combinación `CRTL+C`.

Atributos de StreamTokenizer	
double	nval Si el token actual es un número, esta variable contiene ese número.
String	sval Si el token actual es una palabra, esta variable contiene esa palabra.
static int	TT_EOF Esta constante indica que el final del flujo de datos se ha leído.
static int	TT_EOL Esta constante indica que el final de línea se ha leído.
static int	TT_NUMBER Esta constante indica que el token que se ha leído es un número.
static int	TT_WORD Esta constante indica que el token que se ha leído es una cadena
int	ttype Después de una llamada al método nextToken, este campo contiene el tipo del token que se acaba de leer.

Constructor de StreamTokenizer
StreamTokenizer (InputStream is) Deprecated. <i>Desde JDK version 1.1, la manera preferente para extraer tokens de un stream de bytes es convertirlo a un stream de caracteres, por ejemplo:</i> <pre>Reader r = new BufferedReader(new InputStreamReader(is)); StreamTokenizer st = new StreamTokenizer(r);</pre>
StreamTokenizer (Reader r) Crea un objeto StreamTokenizer a partir de un flujo de caracteres.

Para leer los token se utiliza el método **nextToken()**.

Ej: El siguiente ejemplo es uno de los que usa el tokenizer. Tenemos un fichero de texto con la siguiente información que corresponde a identificativo, nombre y edad de una persona:



```
public class EjStreamTokenizer {
    public static void main(String[] args) throws IOException {
        FileReader r=new FileReader("C:/FicherosJava/Ejemplo.txt");
        StreamTokenizer tokens= new StreamTokenizer(r);
        while (tokens.nextToken() != StreamTokenizer.TT_EOF) {
            if (tokens.ttype == StreamTokenizer.TT_WORD)
                System.out.println(tokens.sval);
            else if (tokens.ttype == StreamTokenizer.TT_NUMBER)
                System.out.println((int) tokens.nval);
        }
    }
}
```

Salida:

```
run:
1
Antonio
4
2
Luis
7
4
Anton
8
BUILD SUCCESSFUL (total time: 0 seconds)
```

9. Serialización

La serialización de un objeto consiste en obtener una secuencia de bytes que represente el estado de dicho objeto. Esta secuencia puede utilizarse de varias maneras (puede enviarse a través de la red, guardarse en un fichero para su uso posterior, utilizarse para recomponer el objeto original, etc.).

Serialización: Posibilidad de escribir/leer Objetos java en Streams.

9.1 Interfaz Serializable

Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes para poder ser tratado por un stream.

Para que un objeto **sea serializable**, debe implementar la interfaz **java.io.Serializable**.

```
import java.io.Serializable;
public class Datos implements Serializable
{
    private int a;
    private String b;
    private char c;
}
```

Esta interfaz no define ningún método. Simplemente se usa para 'marcar' aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruidas). Objetos tan comunes como String, Vector o ArrayList implementan Serializable, de modo que pueden ser serializados y reconstruidos más tarde.

El sistema de ejecución de Java se encarga de hacer la serialización de forma automática.

Este interface **no declara ninguna función miembro**, se trata de un interface vacío. No tiene métodos que debamos redefinir.

```
import java.io.*;
public interface Serializable{
}
```

Solo los objetos que implemente la interfaz Serializable pueden ser escritos a stream. La clase de cada objeto es codificada incluyendo el nombre de la clase y la firma (su prototipo), los valores de sus campos, y cualquier objeto referenciado desde el objeto inicial, o sea, se guarda también una cabecera junto al objeto en el fichero.

Por razones de seguridad, las clases no son serializables por defecto.

9.2 Excluir campos al serializar objetos

Algunas veces es necesario excluir campos a la hora de serializar objetos, por ejemplo cuando se tiene un objeto que guarda la información de un usuario incluida su contraseña.

Para evitar que esos campos sean serializados basta con utilizar el modificador **transient**.

```
private String nombre;
private transient String contrasenia; //Este campo no será guardado
```

9.3 Flujos para la entrada y salida de objetos: ObjectOutputStream e ObjectInputStream

La serialización está orientada a bytes, por lo tanto, se utilizan clases que están en la jerarquía de InputStream y OutputStream..

Estas clases implementan las interface ObjectInput y ObjectOutput, que son subinterfaces de DataInput y DataOutput. Esto significa que todos métodos de E/S para escribir o leer datos en flujos de datos, también se aplican a los flujos para objetos. Por lo tanto un flujo de objeto puede contener una mezcla de valores primitivos y objeto.

Class ObjectOutputStream

```

java.lang.Object
 java.io.OutputStream
  java.io.ObjectOutputStream

All Implemented Interfaces:
  Closeable, DataOutput, Flushable, ObjectOutput, ObjectOutputStreamConstants, AutoCloseable

public class ObjectOutputStream
 extends OutputStream
 implements ObjectOutput, ObjectOutputStreamConstants

```

Class ObjectInputStream

```

java.lang.Object
 java.io.InputStream
  java.io.ObjectInputStream

All Implemented Interfaces:
  Closeable, DataInput, ObjectInput, ObjectStreamConstants, AutoCloseable

public class ObjectInputStream
 extends InputStream
 implements ObjectInput, ObjectStreamConstants

```

- **Para serializar un objeto:** es necesario crear algún objeto de tipo *OutputStream* que se le pasa al constructor de **ObjectOutputStream**. A continuación se puede llamar algún método, por ejemplo, `writeObject()`, para serializar el objeto.

Constructor	Descripción
protected	<code>ObjectOutputStream(OutputStream out)</code> throws IOException Crea un ObjectOutputStream que escribe en el OutputStream especificado.

Ejemplo: Para escribir un objeto en un fichero:

```

FileOutputStream fs=new FileOutputStream(fichero);
ObjectOutputStream os = new ObjectOutputStream(fs);

O de forma abreviada:
ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream(fichero));

```

- **Para recuperar un objeto:** es necesario crear algún objeto de tipo *InputStream* que se le pasa al constructor de **ObjectInputStream**. A continuación se puede llamar algún método, por ejemplo, `readObject()`, para leer el objeto.

Constructor	Descripción
protected	<code>ObjectInputStream(InputStream in)</code> throws IOException Recupera datos primitivos y objetos previamente almacenados con ObjectOutputStream

```

FileInputStream fe=new FileInputStream(fichero);
ObjectInputStream oe = new ObjectInputStream(fe);

O de forma abreviada:
ObjectInputStream oe = new ObjectInputStream(new FileInputStream(fichero));

```

Hay que tener claro el orden y el tipo de los objetos almacenados en el disco para recuperarlos en el mismo orden.

9.4 Escritura de objetos en ficheros

La clase **ObjectOutputStream** permite crear objetos que se asocian a un objeto **FileOutputStream** y facilita métodos para escribir o almacenar secuencialmente información codificada en binario en el fichero asociado a dicho objeto:

Algunos métodos son:

Modifier and Type	Method and Description
void	<code>close()</code> Cierra el flujo de salida.
void	<code>defaultWriteObject()</code> Escribe los campos no estáticos y no transient de la clase actual en el flujo de salida.
protected void	<code>drain()</code> Vacía todos los datos almacenados en el buffer de ObjectOutputStream.
protected boolean	<code>enableReplaceObject(boolean enable)</code> Activa el flujo de salida para la sustitución de objetos en el flujo.
void	<code>flush()</code> Vacía el flujo de salida
protected Object	<code>replaceObject(Object obj)</code>

	Este método permitirá a las subclases de <code>ObjectOutputStream</code> sustituir un objeto por otro durante la serialización.
<code>void</code>	<code>reset()</code> Reset will disregard the state of any objects already written to the stream.
<code>void</code>	<code>write(byte[] buf)</code> Escribe un array de bytes en el flujo de salida
<code>void</code>	<code>write(byte[] buf, int off, int len)</code> Escribe un array de bytes en el flujo de salida a partir del desplazamiento off
<code>void</code>	<code>write(int val)</code> Escribe un byte.
<code>void</code>	<code>writeBoolean(boolean val)</code> Escribe boolean.
<code>void</code>	<code>writeByte(int val)</code> Escribe 1 byte.
<code>void</code>	<code>writeBytes(String str)</code> Escribe un String como secuencia de bytes
<code>void</code>	<code>writeChar(int val)</code> Escribe un char con 16 bit.
<code>void</code>	<code>writeChars(String str)</code> Escribe un String como secuencia de caracteres
<code>protected void</code>	<code>writeClassDescriptor(ObjectStreamClass desc)</code> Escribe el descriptor de la clase especificada en el <code>ObjectOutputStream</code>
<code>void</code>	<code>writeDouble(double val)</code> Writes un double (64 bit)
<code>void</code>	<code>writeFields()</code> Escribe los campos del buffer en el flujo de salida.
<code>void</code>	<code>writeFloat(float val)</code> Escribe un float (32 bit).
<code>void</code>	<code>writeInt(int val)</code> Escribe un int (32 bit).
<code>void</code>	<code>writeLong(long val)</code> Escribe un long (64 bit).
<code>void</code>	<code>writeObject(Object obj)</code> Escribe el objeto especificado en el <code>ObjectOutputStream</code> .
<code>protected void</code>	<code>writeObjectOverride(Object obj)</code> Método usado por las subclases para sobrescribir el método default <code>writeObject</code> .
<code>void</code>	<code>writeShort(int val)</code> Escribe un short (16 bit).
<code>void</code>	<code>writeUTF(String str)</code> Escribe un string con el format UTF-8

Ejemplo: Escritura de objetos punto en un fichero.

```
package ejserializacionobjetosfichero;
import java.io.*;
class punto implements Serializable
{   int x,y;
punto(int x,int y){
    this.x=x;    this.y=y;    }
public int getX(){    return x;}
public int getY(){    return y;}
} //fin clase punto

public class EjSerializacionObjetosFichero {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        ObjectOutputStream salida=
        new ObjectOutputStream (new FileOutputStream("C:/FicherosJava/ejobjetos.dat"));

        salida.writeObject(new punto(5,8));
        salida.writeObject(new punto(1,9));
        salida.writeObject(new punto(3,7));
        salida.close();
        System.out.println("fin de la grabación de objetos");
    }
}
```

9.5 Lectura de objetos en ficheros

La clase **ObjectInputStream** permite crear objetos que se asocian a un objeto **FileInputStream** y facilita métodos para leer de él secuencialmente información codificada en binario:

Algunos métodos son:

Modificador y tipo	Metodo y descripción
int	available() Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	close() Cierra el flujo de entrada
void	defaultReadObject() Lee los campos no estáticos y no transient de la clase actual en el flujo de entrada
protected boolean	enableResolveObject(boolean enable) Activa el flujo de entra para que los objetos leídos puedan ser sustituidos.
int	read() Lee un byte de datos
int	read(byte[] buf, int off, int len) Reads into an array of bytes.
boolean	readBoolean() Lee un boolean.
byte	readByte() Lee un byte (8 bits).
char	readChar() Lee un carácter con 16bits.
protected ObjectStreamClass	readClassDescriptor() Lee el descriptor desde el flujo de serialización.
double	readDouble() Lee un double (64 bit).
float	readFloat() Lee un float (32 bit).
void	readFully(byte[] buf) Lee buf.length bytes y los deposita en el array buf.
void	readFully(byte[] buf, int off, int len) Lee buf.length bytes y los deposita en el array buf a partir del desplazamiento off.
int	readInt() Lee un int (32 bit).
String	readLine() Deprecated.
long	readLong() Lee un long (64 bit).
Object	readObject() Lee un objeto desde el flujo ObjectInputStream.
short	readShort() Lee un short (16 bit).
int	readUnsignedByte() Lee un byte sin signo (8 bit)
int	readUnsignedShort() Lee un short sin signo (16 bit).
String	readUTF() Lee un String en formato UTF-8
int	skipBytes(int len) Salta len bytes.

Ej: Lectura de los objetos punto escrito en el anterior fichero.

```
ObjectInputStream entrada=new ObjectInputStream
    (new FileInputStream("C:/FicherosJava/ejobjetos.dat"));

try
{
    while(true){
        punto puntoentrada=(punto) entrada.readObject();
        System.out.println(" (x="+puntoentrada.getX()+",
                                y="+puntoentrada.getY()+") ");
    }
    //cuando se llega al final se lanza una excepción
    catch (IOException e){
        System.out.println("llegado al final"); }
    entrada.close();
```

Salida:

```

Run:
leyendo el fichero
(x=5, y=8)
(x=1, y=9)
(x=3, y=7)
llegado al final
BUILD SUCCESSFUL (total time: 1 second)

```

9.6 Serialización de objetos compuestos

Si dentro de la clase hay atributos que son otras clases, éstas clases a su vez también deben ser *Serializable*.

Con los tipos de java (String, Integer, etc.) no hay problema porque lo son. Las principales clases en java ya son serializables.

Si ponemos como atributos nuestras propias clases, éstas a su vez deben implementar *Serializable*. Si no java lanza una excepción, por ejemplo: Exception in thread "main" **java.io.NotSerializableException**: ejserializacionficherosobjetoscompuestos.Punto

Ejemplo: la clase Rectangulo tiene un atributo de tipo Punto. La clase Punto debe implementar también la interfaz serializable:

```

package ejserializacionficherosobjetoscompuestos;
import java.io.*;
class Punto implements Serializable
{
    int x,y;
    Punto(int x,int y){
        this.x=x;
        this.y=y;
    }
    public int getX(){ return x;}
    public int getY(){ return y;}
}
class Rectangulo implements java.io.Serializable{
    private int ancho ;
    private int alto ;
    private Punto origen;
    public Rectangulo(int x, int y, int ancho, int alto){
        origen=new Punto(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public int getX(){return origen.getX();}
    }
    public int getY(){return origen.getY();}
    public int getAncho(){return ancho;}
    public int getAlto(){return alto; }
}
public class EjSerializacionFicherosObjetosCompuestos {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        ObjectOutputStream salida=new ObjectOutputStream
            (new FileOutputStream("C:/FicherosJava/ejobjetos2.dat"));
        salida.writeObject(new Rectangulo(3,4,5,8));
        salida.writeObject(new Rectangulo(6,1,5,8));
        salida.writeObject(new Rectangulo(2,4,5,8));
        salida.close();
        System.out.println("leyendo el fichero");
        ObjectInputStream entrada=new ObjectInputStream
            (new FileInputStream("C:/FicherosJava/ejobjetos2.dat"));
        try
        { while(true){
            Rectangulo Rectentrada= (Rectangulo)entrada.readObject();
            System.out.println("(x="+Rectentrada.getX()+",
                y="+Rectentrada.getY()+"), alto="+
                Rectentrada.getAlto()+ " Ancho="+Rectentrada.getAncho());
            }
        }//Si llega al final se produce una excepción IOException
        catch (IOException e){
            System.out.println("llegado al final");
            entrada.close();
        }
    }
}

```

Salida:

```
run:
leyendo el fichero
(x=3, y=4), alto=8 Ancho=5
(x=6, y=1), alto=8 Ancho=5
(x=2, y=4), alto=8 Ancho=5
llegado al final
BUILD SUCCESSFUL (total time: 0 seconds)
```

10. Ficheros de acceso directo

A menudo, no se desea leer un fichero de principio a fin; sino acceder al fichero como una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero.

El paquete **java.io** contiene una clase muy útil para el tratamiento de ficheros de acceso directo, su nombre es **RandomAccessFile** e implementa los interfaces **DataInput** y **DataOutput**, por lo que contiene métodos de lectura y escritura sobre este tipo de ficheros.

La clase **RandomAccessFile** proporciona todos los métodos necesarios para leer y escribir en archivos de manera no necesariamente secuencial, es decir, en forma aleatoria.

- Éstos son los constructores de **RandomAccessFile**:

Constructores:	
RandomAccessFile (File fichero, String modo) throws FileNotFoundException	Abre el fichero de acceso aleatorio especificado en el argumento File para lectura, y opcionalmente, para escritura según se especifique en el argumento modo.
RandomAccessFile (String nombre, String modo) throws FileNotFoundException	Abre el fichero de acceso aleatorio especificado en el argumento nombre para lecturas, y opcionalmente, para escritura según se especifique en el argumento modo.

Modos de acceso:

- **"r"**: Abre el fichero para sólo lectura. La invocación de cualquiera de los métodos de escritura del objeto resultante causará una excepción **IOException** al ser lanzada.
- **"rw"**: Abre el fichero para lectura y escritura. Si el fichero no existe, entonces se intentará crearlo.

Excepciones que se puede lanzar:

- **IllegalArgumentException** - si el argumento mode no es igual a la de "r", "rw".
- **FileNotFoundException** - si el modo de acceso es "r", y el fichero no existe, o si el modo de acceso es "rw", y el fichero no existe, se intenta crear uno nuevo y se produce un error al crearlo, o si existe y no se puede abrir.

Métodos de **RandomAccessFile**:

Modificador y tipo	Métodos y descripción
void	close() throws IOException Cierra el fichero y libera sus recursos asociados.
FileChannel	getChannel() throws IOException Devuelve el objeto único FileChannel asociado con el fichero.
FileDescriptor	getFD() throws IOException Devuelve el objeto descriptor del fichero asociado con el stream.
long	getFilePointer() throws IOException Devuelve la posición actual del punteo de lectura/escritura.
long	length() throws IOException Devuelve la longitud del fichero.
int	read() throws IOException Lee un byte del fichero. Devuelve -1 si no hay ningún byte más que leer.
int	read(byte[] b) throws IOException No lee un solo byte, sino que lee hasta que b.length bytes guardándolos en el array b que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.

int	read (byte[] b, int off, int len) throws IOException Lee hasta len bytes del fichero y los deposita en b a partir de off. Devuelve -1 si no hay ningún byte más que leer.
boolean	readBoolean () throws IOException Lee un boolean desde el fichero. Este método lee un solo byte del archivo, a partir del puntero de archivo actual. Un valor de 0 representa false. Cualquier otro valor representa el verdadero. Lanza EOFException - si el archivo ha llegado al final.
byte	readByte () throws IOException Lee un byte con signo desde el fichero. Lanza EOFException - si el archivo ha llegado al final.
char	readChar () throws IOException Lee un carácter desde el fichero. Lanza una EOFException , si el archivo llega al final antes de leer 2 bytes.
double	readDouble () throws IOException Lee un double desde el fichero. Lanza una EOFException , si el archivo llega al final antes de leer 8 bytes.
float	readFloat () throws IOException Lee un float desde el fichero. Lanza una EOFException , si el archivo llega al final antes de leer 4 bytes.
void	readFully (byte[] b) throws IOException Lee b.length bytes comenzando en la posición actual del puntero de lectura y escritura y los deposita en el array b. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
void	readFully (byte[] b, int off, int len) Lee b.length bytes comenzando en la posición actual del puntero de lectura y escritura y los deposita en el array b a partir de off. Lanza una EOFException , si el archivo llega al final antes de leer todos los bytes.
int	readInt () throws IOException Lee un entero de 32-bit desde el fichero. Lanza una EOFException , si el archivo llega al final antes de leer 4 bytes.
String	readLine () throws IOException Lee una cadena hasta que se encuentre el final de línea desde el fichero.
long	readLong () throws IOException Lee un entero de 64-bit desde el fichero. Lanza una EOFException , si el archivo llega al final antes de leer 4 bytes.
short	readShort () throws IOException Lee un entero corto de 16-bit desde el fichero. Lanza EOFException - si el archivo llega al final antes de la lectura de dos bytes.
int	readUnsignedByte () throws IOException Lee un entero sin signo de 8-bit desde el fichero. Lanza EOFException - si el archivo ha llegado al final.
int	readUnsignedShort () throws IOException Lee un entero corto sin signo de 16-bit desde el fichero. Lanza EOFException - si el archivo llega al final antes de la lectura de dos bytes.
String	readUTF () throws IOException Lee una cadena desde el archivo. La cadena se ha codificado con UTF-8 formato UTF-8 (8-bit Unicode Transformation Format) es un formato de codificación de caracteres Unicode que se usa frecuentemente para transmitir datos y es compatible con ascii). Puede lanzar también EOFException - si el archivo llega al final antes de leer todos los bytes y UTFDataFormatException - si los bytes no representan una cadena válida Unicode con codificación UTF-8.
void	seek (long pos) throws IOException Establece el puntero de lectura y escritura en la posición pos del fichero.
void	setLength (long newLength) throws IOException Establece la longitud del fichero.
int	skipBytes (int n) throws IOException Trata de saltar n bytes en el fichero.
void	write (byte[] b) throws IOException Escribe b.length bytes en el fichero desde el array b en la posición actual del punter de lectura/escritura.
void	write (byte[] b, int off, int len) Escribe len bytes en el fichero desde el array b comenzando en la posición off.
void	write (int b) throws IOException Escribe el primer byte de menor peso en el fichero. Los otros 3 restantes bytes se ignoran.
void	writeBoolean (boolean v) throws IOException Escribe un boolean en el fichero. Si el argumento v es verdadero, el valor (byte) 1 se escribe, y si v es falso, el valor (byte) 0 se escribe. El byte escrito por este método puede ser leído por el readBoolean que devolverá un booleano igual a v.
void	writeByte (int v) throws IOException Escribe el primer byte de menor peso en el fichero. Los otros 3 restantes bytes se ignoran. Hace lo mismo que write(int b).
void	writeBytes (String s) throws IOException Escribe una cadena en el fichero. Por cada carácter de la cadena s, tomadas en orden, un byte se escribe en el fichero.
void	writeChar (int v) throws IOException Escribe un valor char, que se compone de dos bytes, en el fichero.
void	writeChars (String s) throws IOException Escribe una cadena en el fichero. Por cada carácter de la cadena s, tomadas en orden, se escribe un byte en el fichero. Hace lo mismo que writeBytes(String s).
void	writeDouble (double v) throws IOException Escribe un numero real Double.
void	writeFloat (float v) throws IOException Escribe un número real Float.
void	writeInt (int v) throws IOException Escribe un número entero.
void	writeLong (long v) throws IOException Escribe un número entero long
void	writeShort (int v) throws IOException Escribe un numero entero Short.

```
void writeUTF(String str) throws IOException
Escribe una cadena UNICODE utilizando codificación UTF-8.
```

Ejemplo:

```
package ejficheroaccesoaleatorio;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
/* @author María José Galán
 */
public class EjFicheroAccesoAleatorio {
    public static void main(String[] args) throws IOException {
        RandomAccessFile f=null;
        try{
            //si el fichero no existe, se crea
            f=new RandomAccessFile("C:/FicherosJava/ejemplo1.txt","rw");
            //siturarse al final del archivo
            f.seek(f.length());
            String s="esto se va a añadir al final del archivo\n";
            f.writeBytes(s);
            //situar el puntero de L/E al principio del archivo.
            f.seek(0L);
            String cad=f.readLine();
            while (cad!=null){
                System.out.println(cad);
                cad=f.readLine();
            }
        }
        catch (FileNotFoundException e){
            System.out.println("Error al abrir el archivo");
        }
        finally
        {
            if (f!=null) f.close();
        }
    }
}
```

Salida:

```
run:
primera linea
segunda linea
tercera linea
cuarta linea
esto se va a añadir al final del archivo
BUILD SUCCESSFUL (total time: 0 seconds)
```