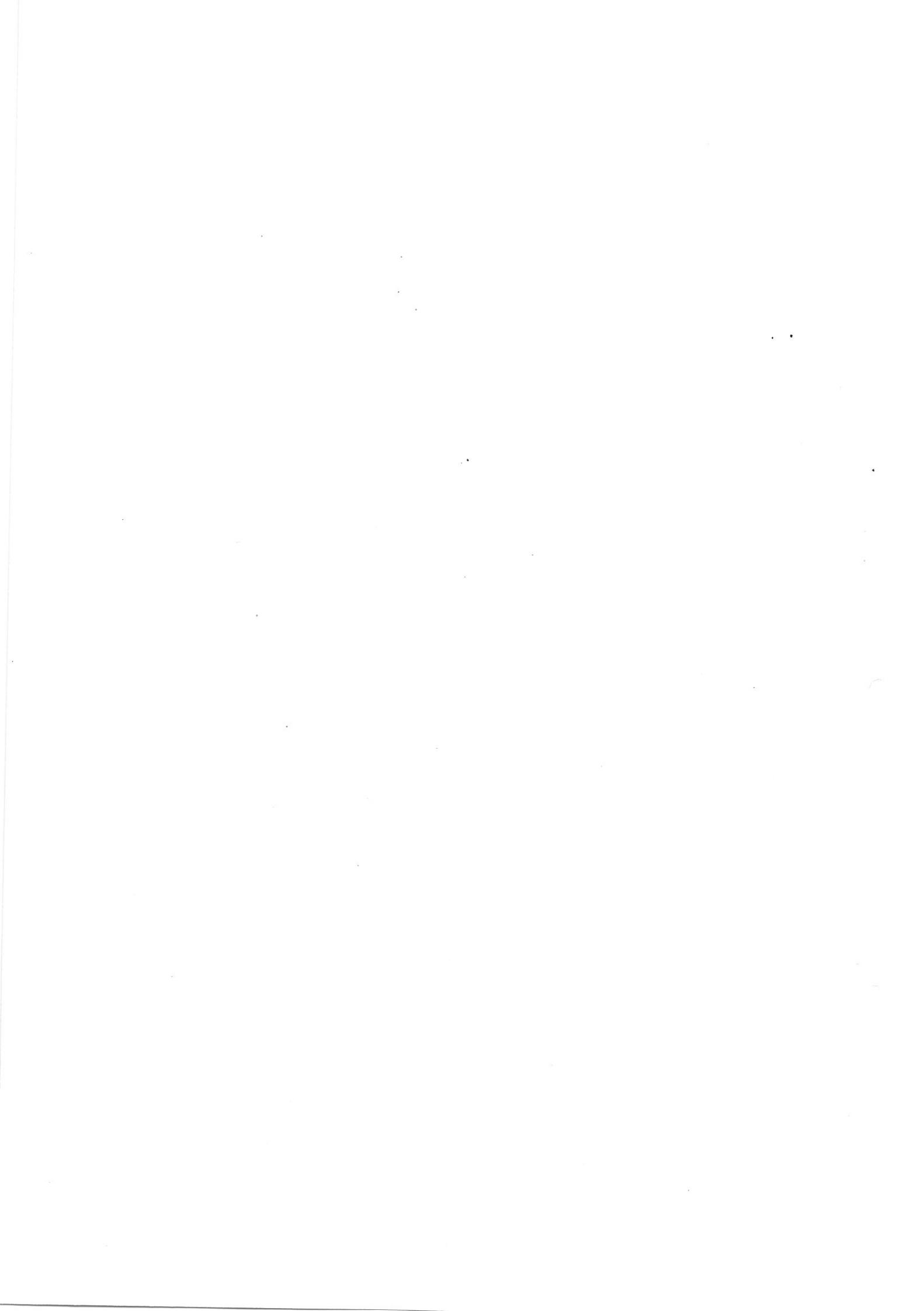




*Implementar un main para:*

- Crear un objeto de la clase concesionario que contenga un ArrayList de objetos coche. Cada objeto del ArrayList se debe corresponder con un registro de la tabla coches de la base de datos concesionario.
- Actualizar aquellos objetos de dicho ArrayList cuyo atributo color sea "ROJO" cambiándolo por "MAGENTA".
- Insertar dos nuevos objetos coche en el ArrayList con los datos necesarios. (Pueden ser inventados).
- Volver a guardar los datos del ArrayList en la tabla coches, para mantener la persistencia de la información.
- Por último, hacer una consulta sobre la base de datos y mostrarla en pantalla. El objetivo es comprobar que los cambios se han guardado correctamente y que ahora hay 12 registros en la base de datos.







clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que un objeto sea serializable.

La clase **ObjectInputStream** y la clase **ObjectOutputStream** se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de **InputStream** y **OutputStream**, de hecho son casi iguales a **DataInput/OutputStream** sólo que incorporan los métodos **readObject** y **writeObject** que son los que permiten grabar directamente objetos. Ejemplo:

```
try{  
    FileInputStream fos=new FileInputStream("d:/nuevo.out");  
    ObjectInputStream os=new ObjectInputStream(fos);  
    Coche c;  
    boolean finalArchivo=false;  
  
    while(!finalArchivo){  
        c=(Coche) readObject(); //Casting necesario porque devuelve Object  
        System.out.println(c);  
    }  
}  
catch(EOFException e){  
    System.out.println("Se alcanzó el final");  
}  
catch(ClassNotFoundException e){  
    System.out.println("Error el tipo de objeto no es compatible");  
}  
catch(FileNotFoundException e){  
    System.out.println("No se encontró el archivo");  
}  
catch(IOException e){  
    System.out.println("Error "+e.getMessage());  
    e.printStackTrace();  
}
```

El listado anterior podría ser el código de lectura de un archivo que guarda coches. Los métodos **readObject** y **writeObject** usan objetos de tipo **Object**, **readObject** les devuelve y **writeObject** les recibe como parámetro. Ambos métodos lanzan excepciones del tipo **IOException** y **readObject** además lanza excepciones del tipo **ClassNotFoundException**.

Obsérvese en el ejemplo como la excepción **EOFException** ocurre cuando se alcanzó el final del archivo al igual que ocurre con las corrientes binarias de datos

Ejercicio nº 55

- Pdf Adjunto.

(24)

Ejercicio nº 56

- Pdf Adjunto. Llamadas.





### (10.7.2) lectura en archivos binarios

El proceso es análogo. Sólo que hay que tener en cuenta que al leer se puede alcanzar el final del archivo. Al llegar al final del archivo, se produce una excepción del tipo **EOFException** (que es subclase de **IOException**), por lo que habrá que controlarla.

Ejemplo, leer el contenido del archivo del ejemplo anterior:

```
boolean finArchivo=false;//Para provocar bucle infinito
try{
    FileInputStream fis=new FileInputStream(f);
    DataInputStream dis=new DataInputStream(fis);
    double d;
    while (!finArchivo){
        d=dis.readDouble();
        System.out.println(d);
    }
    dis.close();
}
catch(EOFException e){
    finArchivo=true;
}
catch(FileNotFoundException e){
    System.out.println("No se encontró el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
```

En este listado, obsérvese como el bucle **while** que da lugar a la lectura se ejecuta indefinidamente (no se pone como condición a secas **true** porque casi ningún compilador lo acepta), se saldrá de ese bucle cuando ocurra la excepción **EOFException** que indicará el fin de archivo.

Para manejar adecuadamente los archivos hay que conocer perfectamente su contenido.

### (10.8) archivos de acceso aleatorio

Hasta ahora los archivos se están leyendo secuencialmente. Es decir desde el inicio hasta el final. Pero es posible leer datos de una zona concreta del archivo.

Por supuesto esto implica necesariamente dominar la estructura del archivo, pero además permite crear programas muy potentes para manejar archivos de datos binarios.



Por ejemplo este el código de un programa que lee por teclado texto hasta que el usuario deja vacía la línea y todo lo escrito lo vuelca en un archivo llamado **salida.txt**:

```
File f=new File("d:\\salida.txt");
try {
    FileWriter fw=new FileWriter(f);
    BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
    String texto=" ";
    while(texto.length()>0){
        texto=br.readLine();
        fw.write(texto+"\r\n");
    }
    fw.close();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Para leer se utiliza el método **read** que devuelve un **int** y que puede recibir un array de caracteres en el que se almacenarán los caracteres leídos. Ambos métodos pueden provocar excepciones de tipo **IOException**.

No obstante sigue siendo un método todavía muy rudimentario. Por ello lo ideal es convertir el flujo de las clases **File** en clases de tipo **BufferedReader** y **BufferedWriter** vistas anteriormente . Su uso sería:

```
File f=new File("D:/salida.txt");
int x=105;
try{
    FileReader fr=new FileReader(f);
    BufferedReader br=new BufferedReader(fr);
    String s="";
    br.readLine();
    while(s!=null){
        System.out.println(s);
        s=br.readLine();
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

Este código muestra el archivo por pantalla. **readLine** devuelve **null** cuando se llega al final del archivo. Sin embargo es conveniente adelantar la primera lectura antes del bucle para conseguir que no salga por pantalla el propio null (a esto se le llama bucle de lectura adelantada).

Ejemplo:

```
String texto="";
try{
    //Obtención del objeto Reader
    InputStreamReader conv=new InputStreamReader(System.in);
    //Obtención del BufferedReader
    BufferedReader entrada=new BufferedReader(conv);
    texto=entrada.readLine();
}
catch(IOException e){
    System.out.println("Error");
}
System.out.println(texto);
```

## (10.5) lectura y escritura en archivos

### (10.5.1) clases FileInputStream y FileOutputStream

Se trata de las clases que manipulan archivos. Son herederas de **Input/OutputStream**, por lo que manejan corrientes de datos en forma de bytes binarios. La diferencia es que se construyen a partir de objetos de tipo **File**.

### (10.5.2) lectura y escritura byte a byte de un archivo

Para leer necesitamos un archivo del que dispongamos permisos de escritura y su ruta o bien un objeto **File** que le haga referencia. Con ello creamos una corriente de tipo **FileInputStream**:

```
FileInputStream fis=new FileInputStream(objetoFile);
FileInputStream fos=new FileInputStream("/textos/texto25.txt");
```

La construcción de objetos **FileOutputStream** se hace igual, pero además se puede indicar un parámetro más de tipo booleano que con valor **true** permite añadir más datos al archivo (normalmente al escribir se borra el contenido del archivo, valor **false**).

Estos constructores intentan abrir el archivo, generando una excepción del tipo **FileNotFoundException** si el archivo no existiera u ocurriera un error en la apertura. Los métodos de lectura y escritura de estas clases son los heredados de las clases **InputStream** y **OutputStream**; fundamentalmente los métodos **read** y **write** son los que permiten leer y escribir. El método **read** devuelve **-1** en caso de llegar al final del archivo.

Este método lee el archivo de forma absolutamente binaria los archivos y sólo es válido cuando deseamos leer toda la información del archivo.

```
        buffer.append((char)ch);
    }
    in.close();
    return buffer.toString();
}
catch (IOException e) {
    e.printStackTrace();
    return null;
}
}
```

El método principal del programa StreamConverter invoca el método writeOutput para crear un archivo de bytes codificados en UTF-8. El método readInput lee el mismo archivo, convirtiendo los bytes de nuevo en Unicode. Aquí está el código fuente para el método main():

```
public static void main(String[] args) {
    String jaString = new
String("\u065e\u0672c\u08a9e\u06587\u05b57\u05217");
    writeOutput(jaString);
    String inputString = readInput();
    String displayString = jaString + " " + inputString;
    new ShowString(displayString, "Conversion Demo");
}
```

La cadena original (jaString) debe ser idéntica a la cadena recién creada (inputString). Para demostrar que las dos cadenas son iguales, el programa les concatena y las muestra con un objeto ShowString.

salida estándar (normalmente la pantalla). Hay también una clase **err** que representa a la salida estándar para errores. El uso podría ser:

```
InputStream stdin =System.in;
OutputStream stdout=System.out;
```

El método **read()** permite leer un byte. Este método puede lanzar excepciones del tipo **IOException** por lo que debe ser capturada dicha excepción.

```
int valor=0;
try{
    valor=System.in.read();
}
catch(IOException e){
    ...
}
System.out.println(valor);
```

No tiene sentido el listado anterior, ya que **read()** lee sólo un byte de la entrada estándar (del teclado), y lo normal es escribir textos largos; por lo que el método **read** no es el apropiado. El método **read** puede poseer un argumento que es un array de bytes que almacenará cada carácter leído y devolverá el número de caracteres leído

```
InputStream stdin=System.in;
int n=0;
byte[] caracter=new byte[1024];
try{
    n=System.in.read(caracter);
}
catch(IOException e){
    System.out.println("Error en la lectura");
}
for (int i=0;i<=n;i++)
    System.out.print((char)caracter[i]);
```

El lista anterior lee una serie de bytes y luego los escribe. La lectura almacena el código del carácter leído, por eso hay que hacer una conversión a **char**.

Para saber que tamaño dar al array de bytes, se puede usar el método **available** de la clase **InputStream** la tercera línea del código anterior sería:

```
byte[] carácter=new byte[System.in.available()];
```

#### (10.4.2) conversión a forma de Reader

El hecho de que las clases **InputStream** y **OutputStream** usen el tipo **byte** para la lectura, complica mucho su uso. Desde que se impuso Unicode y con él las

### (10.3.1) DataInputStream/DataOutputStream

Leen corrientes de datos de entrada en forma de byte, pero adaptándola a los tipos simples de datos (`int`, `short`, `byte`, ..., `String`). Tienen varios métodos `read` y `write` para leer y escribir datos de todo tipo.

Ambas clases construyen objetos a partir de corrientes `InputStream` y `OutputStream` respectivamente.

#### métodos de DataInputStream

método	uso
<code>boolean readBoolean()</code>	Lee un valor booleano de la corriente de entrada. Puede provocar excepciones de tipo <code>IOException</code> o excepciones de tipo <code>EOFException</code> , esta última se produce cuando se ha alcanzado el final del archivo y es una excepción derivada de la anterior, por lo que si se capturan ambas, ésta debe ir en un catch anterior (de otro modo, el flujo del programa entraría siempre en la <code>IOException</code> ).
<code>byte readByte()</code>	Idéntica a la anterior, pero obtiene un byte. Las excepciones que produce son las mismas
<code>... readChar(), readShort(), readLong(), readFloat(), readDouble()</code>	Como las anteriores pero devolviendo el tipo de datos adecuado
<code>String readLine()</code>	Lee de la entrada caracteres hasta llegar a un salto de línea o al fin del fichero y el resultado lo obtiene en forma de <code>String</code>
<code>String readUTF()</code>	Lee un <code>String</code> en formato UTF (codificación norteamericana). Además de las excepciones comentadas antes, puede ocurrir una excepción del tipo <code>UTFDataFormatException</code> (derivada de <code>IOException</code> ) si el formato del texto no está en UTF.

#### métodos de OutputStreamWriter

La idea es la misma, los métodos son: `writeBoolean`, `writeByte`, `writeBytes` (para `Strings`), `writeFloat`, `writeShort`, `writeUTF`, `writeln`, `writeInt`, `writeLong`. Todos poseen un argumento que son los datos a escribir (cuyo tipo debe coincidir con la función).

### (10.3.2) ObjectInputStream/ObjectOutputStream

Filtros de secuencia que permiten leer y escribir objetos de una corriente de datos orientada a bytes. Sólo tiene sentido si los datos almacenados son objetos. Tienen los mismos métodos que la anterior, pero aportan un nuevo método de lectura:

### métodos de OutputStream

método	uso
<code>void close()</code>	Cierra la corriente de salida. Cualquier acceso posterior generaría una <b>IOException</b> .
<code>void flush()</code>	Vacia los búferes de salida de la corriente de datos
<code>void write(int byte)</code>	Escribe un byte en la corriente de salida
<code>void write(byte[] bufer)</code>	Escribe todo el array de bytes en la corriente de salida
<code>void write( byte[] buffer, int posInicial, int numBytes )</code>	Escribe el array de bytes en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por <b>numBytes</b> .

### (10.3.2) Reader/Writer

Clases abstractas que definen las funciones básicas de escritura y lectura basada en texto Unicode. Se dice que estas clases pertenecen a la jerarquía de lectura/escritura orientada a caracteres, mientras que las anteriores pertenecen a la jerarquía orientada a bytes.

Aparecieron en la versión 1.1 y no substituyen a las anteriores. Siempre que se pueda es más recomendable usar clases que deriven de estas.

Poseen métodos **read** y **write** adaptados para leer arrays de caracteres.

### métodos de Reader

método	uso
<code>void close()</code>	Cierra la corriente de entrada. Cualquier acceso posterior generaría una <b>IOException</b> .
<code>void mark(int bytes)</code>	Marca la posición actual en el flujo de datos de entrada. Cuando se lea el número de bytes indicado, la marca se elimina.
<code>boolean markSupported()</code>	Devuelve verdadero si en la corriente de entrada es posible marcar mediante el método <b>mark</b> .
<code>int read()</code>	Lee el siguiente byte de la corriente de entrada y le almacena en formato de entero. Devuelve -1 si estamos al final del fichero
<code>int read(byte[] búfer)</code>	Lee de la corriente de entrada bytes y les almacena en el búfer. Lee hasta llenar el búfer.
<code>int read( byte[] bufer, int posInicial, int despl)</code>	Lee de la corriente de entrada bytes y les almacena en el búfer. La lectura la almacena en el array pero a partir de la posición indicada, el número máximo de bytes leídos es el tercer parámetro.
<code>boolean ready()</code>	Devuelve verdadero si la corriente de entrada está lista.

## - Coniertos de Bytes

- Stream → conierte byte a byte

~~Stream~~

- caracteres (dos bytes)

- Excepciones derivadas todas de IOException.  
(Extiende de exception que es la  
en el paquete java.io.)

- Coniertos de bytes. InputStream / OutputStream

Stream  
- Eslos dos clases de linea los flujo  
binarios de lectura / escritura. Se usa  
secuencia de bytes. (son datos binaria-  
rios propios)  
- Metodos mas importantes ready()  
unread().

- Reader / writer, clases para almacenar  
y resguardar como texto Unicode.

## (10.3) clases para la entrada y la salida

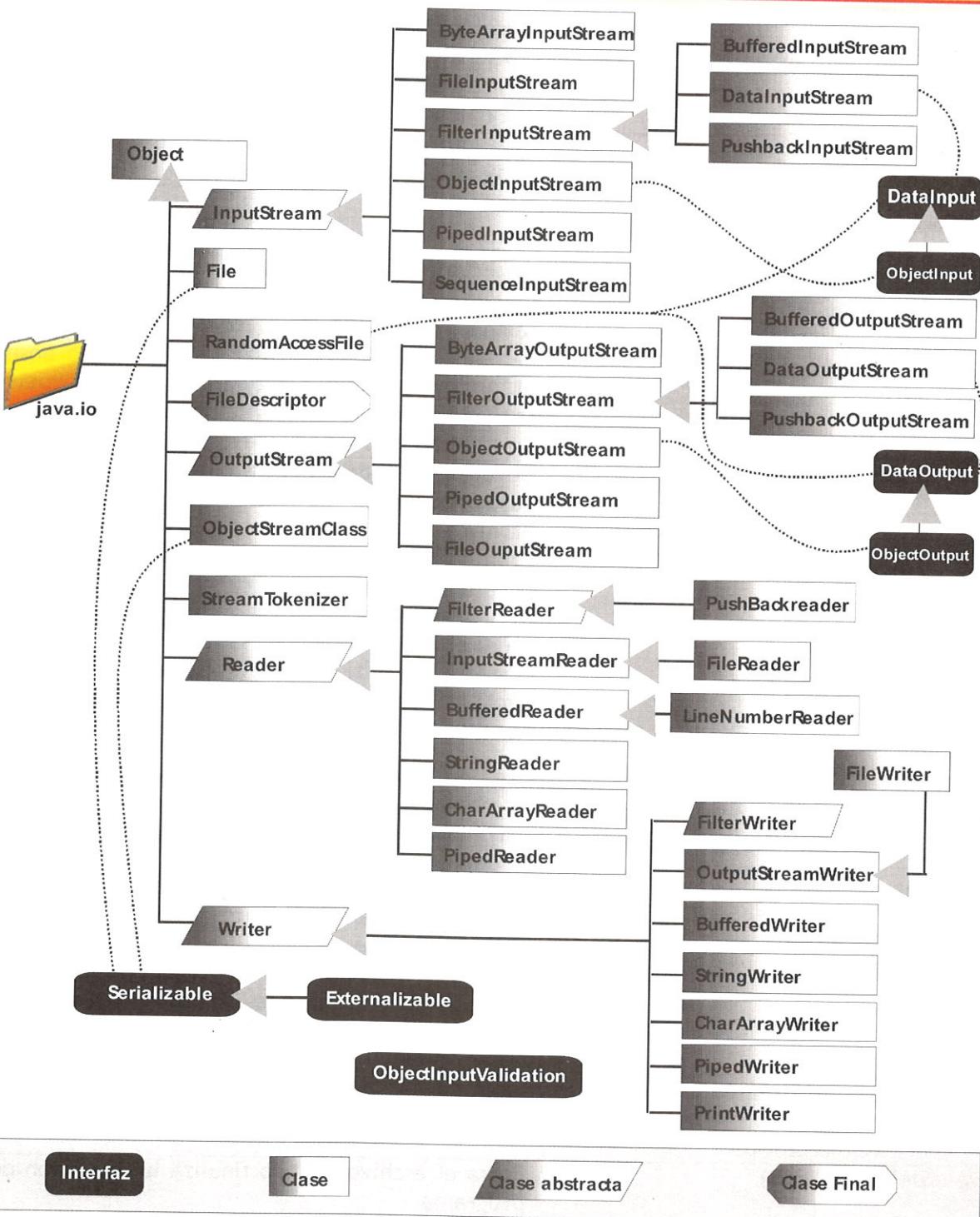


Ilustración 10-1, Clases e interfaces del paquete `java.io`

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

Normalmente no es necesaria esta comprobación ya que Windows acepta también el carácter / como separador.

### métodos generales

método	uso
<code>String toString()</code>	Para obtener la cadena descriptiva del objeto
<code>boolean exists()</code>	Devuelve <code>true</code> si existe la carpeta o archivo.
<code>boolean canRead()</code>	Devuelve <code>true</code> si el archivo se puede leer
<code>boolean canWrite()</code>	Devuelve <code>true</code> si el archivo se puede escribir
<code>boolean isHidden()</code>	Devuelve <code>true</code> si el objeto File es oculto
<code>boolean isAbsolute()</code>	Devuelve <code>true</code> si la ruta indicada en el objeto File es absoluta
<code>boolean equals(File f2)</code>	Compara <code>f2</code> con el objeto <code>File</code> y devuelve verdadero si son iguales.
<code>int compareTo(File f2)</code>	Compara basado en el orden alfabético del texto (sólo funciona bien si ambos archivos son de texto) <code>f2</code> con el objeto <code>File</code> y devuelve cero si son iguales, un entero negativo si el orden de <code>f2</code> es mayor y positivo si es menor
<code>String getAbsolutePath()</code>	Devuelve una cadena con la ruta absoluta al objeto <code>File</code> .
<code>File getAbsoluteFile()</code>	Como la anterior pero el resultado es un objeto <code>File</code>
<code>String getName()</code>	Devuelve el nombre del objeto <code>File</code> .
<code>String getParent()</code>	Devuelve el nombre de su carpeta superior si la hay y si no <code>null</code>
<code>File getParentFile()</code>	Como la anterior pero la respuesta se obtiene en forma de objeto <code>File</code> .
<code>boolean setReadOnly()</code>	Activa el atributo de sólo lectura en la carpeta o archivo.
<code>URL toURL() throws MalformedURLException</code>	Convierte el archivo a su notación URL correspondiente
<code>URI toURI()</code>	Convierte el archivo a su notación URI correspondiente

### métodos de carpetas

método	uso
<code>boolean isDirectory()</code>	Devuelve <code>true</code> si el objeto <code>File</code> es una carpeta y <code>false</code> si es un archivo o si no existe.
<code>boolean mkdir()</code>	Intenta crear una carpeta y devuelve <code>true</code> si fue posible hacerlo
<code>boolean mkdirs()</code>	Usa el objeto para crear una carpeta con la ruta creada para el objeto y si hace falta crea toda la estructura de carpetas necesaria para crearla.

## (10.1) entrada y salida en Java

El paquete **java.io** contiene todas las clases relacionadas con las funciones de entrada (**input**) y salida (**output**). Se habla de E/S (o de I/O) refiriéndose a la entrada y salida. En términos de programación se denomina **entrada** a la posibilidad de introducir datos hacia un programa; **salida** sería la capacidad de un programa de mostrar información al usuario.

Todas las clases relacionadas con la entrada y salida de datos están en el paquete **java.io**.

## (10.2) archivos

### (10.2.1) introducción

Todas las estructuras comentadas en los temas anteriores para almacenar los datos, residen en memoria. De hecho cuando el programa termina, desaparecen.

Para evitar este problema existen los archivos. Se trata de un elemento que pone a disposición el Sistema Operativo para almacenar información de todo tipo bajo un **nombre** y una **extensión**. La extensión indica el tipo de archivo que tenemos.

En definitiva es la estructura habitual en la que se almacena información y cuyo manejo parte de que sabemos sus fundamentos en base al sistema operativo que estemos manejando. Su finalidad es la de poder almacenar datos de forma permanente.

### (10.2.2) clase File

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.

#### construcción de objetos de archivo

Utiliza como único argumento una cadena que representa una ruta en el sistema de archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

```
File archivo1=new File("/datos/bd.txt");
File carpeta=new File("datos");
```



