2 OPERACIONES DE GESTIÓN DE FICHEROS

Las operaciones básicas que se realizan sobre cualquier fichero, independientemente de la forma de acceso al mismo, son las siguientes:

- **Creación del fichero.** El fichero se crea en el disco con un nombre que después se debe usar para acceder a él. Este proceso solo se hace una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, primero debe realizar su apertura. Para ello, el programa utilizará algún método para identificar el fichero, como usar un descriptor del fichero.
- **Cierre del fichero.** El fichero se debe cerrar cuando el programa no lo vaya a utilizar. Suele ser la última instrucción del programa.
- **Lectura de datos del fichero.** Este proceso consiste en transferir información del fichero a la memoria, a través de variables del programa.
- **Escritura de datos en el fichero**. Este proceso consiste en transferir información de la memoria, por medio de las variables del programa, al fichero.

Una vez abierto, las operaciones típicas que se realizan sobre un fichero son:

- **Altas**: consiste en añadir un nuevo registro al fichero.
- **Bajas**: consiste en eliminar del fichero un registro ya existente.
- **Modificaciones**: consiste en cambiar parte del contenido de un registro. Antes de la modificación, será necesario localizar el registro a modificar dentro del fichero.
- **Consultas**: consiste en buscar en el fichero un registro determinado.

2.1 Clases asociadas con ficheros

En Java, la clase **`File`** proporciona un conjunto de utilidades relacionadas con ficheros que proporcionan información acerca de los mismos. Un objeto de la clase **`File`** puede representar el nombre de un fichero o de un directorio que existe en el sistema de ficheros. También se puede usar para crear un nuevo directorio o una trayectoria de directorios completa si ésta no existe.

Para crear un objeto **`File`**, se puede usar cualquiera de los siguientes constructores:

Constructor	Explicación
`File(String directorioYFichero)`	Recibe como parámetro el camino completo donde está el fichero junto con el nombre. Por defecto, si no se indica, lo busca en la carpeta del proyecto. `directorioYFichero` puede ser también el camino a un directorio, sin indicar al final el nombre de ningún fichero. En este caso se crea un directorio.
`File(String directorio, String nombreFichero)`	Recibe en la instanciación del objeto el camino completo donde está el fichero, como primer parámetro, y el nombre del fichero como segundo parámetro.
`File(File directorio, String fichero)`	Recibe en la instanciación del objeto un objeto de tipo `File`, que hace referencia a un directorio, como primer parámetro y el nombre del fichero como segundo parámetro.

La clase **`File`** tiene los siguientes métodos que sirven para ficheros y directorios:

Método	Explicación
`boolean canRead()`	Informa si se puede leer la información que contiene.
`boolean canWrite()`	Informa si se puede guardar información.
`boolean exists()`	Informa si el fichero o el directorio existe.
`boolean isFile()`	Devuelve true si el objeto File corresponde a un fichero normal.
`boolean isDirectory()`	Devuelve true si el objeto `File` corresponde a un directorio.
`long lastModified()`	Retorna la fecha de la última modificación.
`String getName()`	Devuelve el nombre del fichero o directorio.
`String getPath()`	Devuelve la ruta relativa.
`String getAbsolutePath()`	Devuelve la ruta absoluta del fichero/directorio.
`String getParent()`	Devuelve el nombre del directorio padre o null si no existe.

Algunos de los métodos que tiene la clase **`File`**, exclusivos de manejo de ficheros, son:

Método	Explicación
`boolean delete()`	Borra el fichero o directorio asociado al objeto `File`.
`long length()`	Retorna el tamaño del archivo en bytes.
`boolean renameTo(File nuevo_nombre)`	Cambia el nombre del fichero.

Algunos de los métodos que tiene la clase **`File`**, exclusivos de manejo de directorios, son:

Método	Explicación
`boolean mkdir()`	Crea un directorio con el nombre que se haya indicado en el constructor al crear el objeto `File`. Sólo lo crea si no existe.
`String[] list()`	Devuelve un array de String con los nombres de ficheros y directorios asociados al objeto `File`.
`File[] listFiles()`	Devuelve un array de objetos File que corresponden a los a los objetos `File` que están dentro del directorio representado por el objeto `File`.

Hay que tener en cuenta que se puede:

- indicar el nombre de un fichero **sin la ruta**, se buscará el fichero en el directorio actual.
- indicar el nombre de un fichero con la **ruta relativa**.
- indicar el nombre de un fichero con la **ruta absoluta**.

El siguiente programa Java muestra los nombres de los archivos y directorios que se encuentren en el directorio que se pasa como argumento a **`metodo`**.

```
import java.io.File;

public class _2x2x01
{
    public static void metodo(String[] args)
    {
        File file = new File(args[0]);

        if( file.isDirectory() )
        {
            File[] ficheros = file.listFiles();

            for( File f : ficheros )
                System.out.println(f.getName());
        }
    }

    public static void main(String[] args)
    {
        String[] argumentos = { "C:\\" };
        metodo(argumentos);
    }
}
```

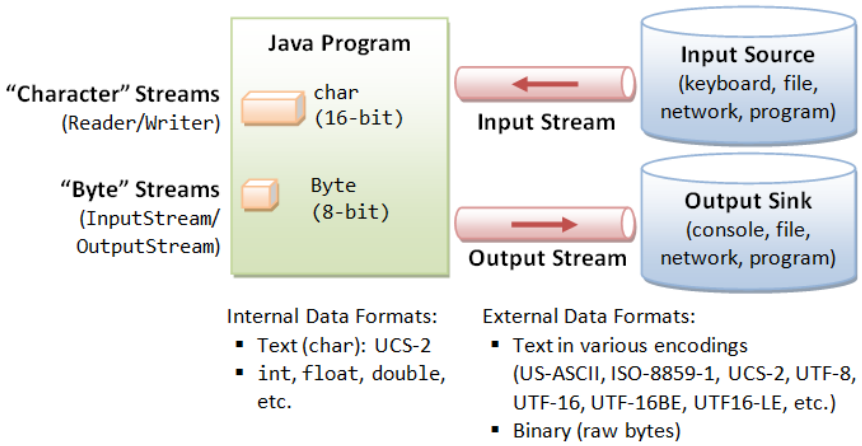
3 ACCESO SECUENCIAL

El sistema de entrada/salida en Java presenta una gran variedad de clases que se implementan en el paquete **`java.io`**. Utiliza la abstracción del flujo o **`stream`** para tratar la comunicación de información entre una fuente y un destino, y por tanto, las operaciones que un programa realiza sobre un flujo son independientes del dispositivo al que esté asociado.

Así pues, un **archivo** es simplemente un flujo externo, es decir, una secuencia de bytes almacenados en un dispositivo externo. Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo o a otro.

Hay dos tipos de flujos de datos definidos:

- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases **`Reader`** y **`Writer`**.
- **Flujos de bytes (8 bits):** realizan operaciones de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. Todas las clases de flujos de bytes heredan de las clases **`InputStream`** y **`OutputStream`**.



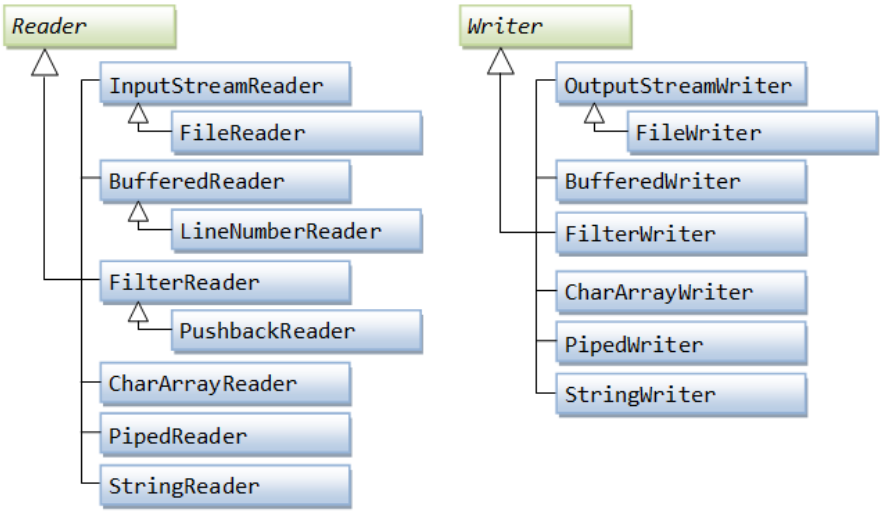
En Java, la entrada desde el teclado y la salida por la pantalla están gestionadas por la clase **`System`**. Esta clase pertenece al paquete **`java.lang`** y tienen tres atributos, los llamados flujos predefinidos: **`in`**, **`out`** y **`err`**, que son **`public`** y **`static`**:

- **`System.in`**: hace referencia a la entrada estándar de datos (teclado).
- **`System.out`**: hace referencia a la salida estándar de datos (pantalla).
- **`System.err`**: hace referencia a la salida estándar de información de errores (pantalla).

3.1 FLUJOS DE CARACTERES

Las clases abstractas **`Reader`** y **`Writer`** manejan flujos de caracteres `Unicode`. Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto, existen varias clases "puente" (que convierten los flujos de bytes a flujos de caracteres): **`InputStreamReader`** convierte un **`InputStream`** en un **`Reader`** (lee bytes y los convierte a caracteres) y **`OutputStreamWriter`** convierte un **`OutputStream`** en un **`Writer`**.

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de caracteres.



Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **`FileReader`** y **`FileWriter`**.
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres: **`CharArrayReader`** y **`CharArrayWriter`**.
- Para buferización de datos, **`BufferedReader`** y **`BufferedWriter`**, las cuales se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que usan un buffer intermedio entre la memoria y el flujo de datos.

3.1.1 Las Clases FileReader y FileWriter

Los ficheros de texto, que normalmente se generan con un editor de textos, almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF-8, etc.). Para trabajar con ellos en Java, se utilizan la clase **`FileReader`** para leer caracteres y la clase **`FileWriter`** para escribir los caracteres en el fichero. Además, las lecturas o escrituras realizadas sobre un fichero deben escribirse dentro de un manejador de excepciones try-catch.

Al instanciar un objeto con la clase **`FileReader`**, el programa abre el fichero que se envía como argumento para lectura y su información se puede leer de forma secuencial carácter a carácter. Si el fichero no existe o no es válido, se lanza la excepción FileNotFoundException.

Constructores de la clase `FileReader`:

Método	Explicación
`FileReader(File fichero)`	Lanza la excepción `FileNotFoundException` si el fichero pasado no existe.
`FileReader(String fichero)`	Lanza la excepción `FileNotFoundException` si el fichero pasado no existe.

Principales métodos de la clase FileReader:

Método	Explicación
`int read()`	Lee un carácter y lo devuelve.
`int read(char[] buf)`	Lee hasta buf.length caracteres de datos del array buf pasado como parámetro.
`int read(char[] buf, int desplazamiento, int n)`	Lee hasta n caracteres de datos del array buf comenzando por buf[desplazamiento] y devuelve el número leído de caracteres.

Al instanciar un objeto con la clase **`FileWriter`**, el programa abre el fichero especificado con el fin de guardar información, pero carácter a carácter. Si el fichero no existe, el programa lo crea de forma automática. Si el disco está lleno o protegido contra escritura, se lanza la excepción **`IOException`**.

Constructores de la clase FileWriter:

Constructor	Explicación
`FileWriter(String nombreFich)`	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileWriter(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileWriter(String nombreDeFich, boolean append)`	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
`FileWriter(File fichero, boolean append)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase FileWriter:

Método	Explicación
`void write(int c)`	Escribe un carácter.
`void write(char[] buf)`	Escribe un array de caracteres.
`void write(char[] buf, int desplazamiento, int n)`	Escribe n caracteres de datos de un array buf comenzando por buf[desplazamiento].
`void write(String str)`	Escribe una cadena de caracteres.
`append(char c)`	Añade un carácter a un fichero.

```
import java.io.*;
public class Ejemplo01 {

	public static void main(String args[]) throws IOException{
		String cadena;
		int codCaracter;
		char caracter;

		FileWriter fichEsc = new FileWriter("nuevo.txt");
		cadena = Leer.pedirCadena("\nIntroduce una frase: ");
		fichEsc.write(cadena);
		fichEsc.close();

		FileReader fichLect = new FileReader("nuevo.txt");
		cadena = "";
		codCaracter = fichLect.read();
		while(codCaracter!=-1){
			caracter = (char) codCaracter;
			cadena = cadena + caracter;
			codCaracter = fichLect.read();
		}
		fichLect.close();
		System.out.println("\nLa frase leída del fichero es: \"" + cadena + "\"");
	}
  }
```

3.1.2 Las Clases BufferedReader y BufferedWriter

Las clases **`BufferedReader`** y **`BufferedWriter`** se pueden utilizar sobre las clases **`FileReader`** y **`FileWriter`** u otros flujos de caracteres para realizar operaciones de entrada/salida con un buffer, en lugar de carácter a carácter.

La clase **BufferedReader** dispone del método **readLine**(), que lee una línea del fichero y la devuelve, o devuelve **null** si no hay nada que leer o se llega al final del fichero. Para construir un objeto **BufferedReader**, se necesita la clase **FileReader**:

```
FileReader fileR = new FileReader(nombreFichero);
BufferedReader bufferedR = new BufferedReader(fileR);
```

```
import java.io.*;
public class Ejemplo02 {

	public static void main(String[] args) throws IOException{
		String cadena;
		File fich = new File("fichero.txt");
		BufferedReader flujo = new BufferedReader(new FileReader(fich));
		if (fich.exists()){
			System.out.println("\nEsta es la información que contiene el
					fichero: ");
			cadena = flujo.readLine();
			while(cadena!=null){
				System.out.println(cadena);
				cadena = flujo.readLine();
			}
		}
	}
}
```

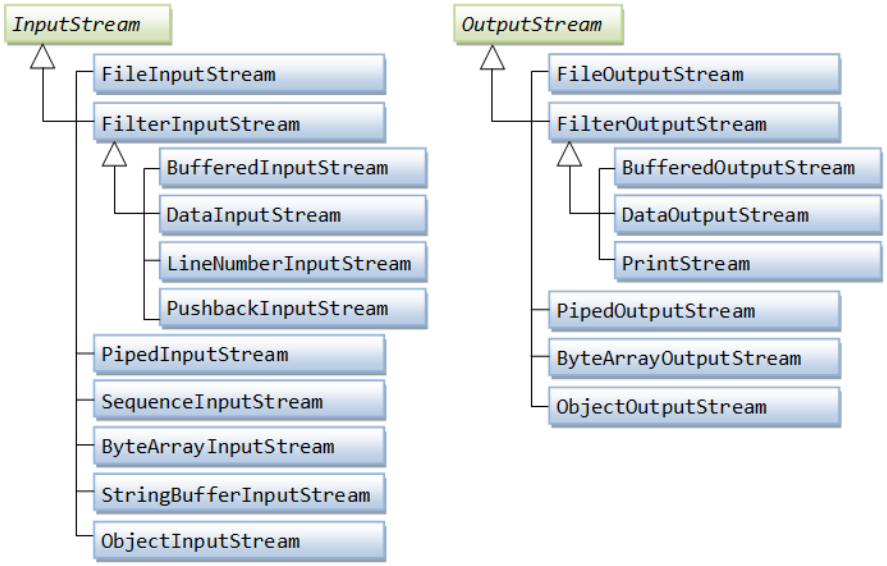
La clase **BufferedWriter** dispone del método **write**(), que escribe una línea en el fichero, y del método **newLine**(), que escribe un salto de línea en el fichero. Para construir un objeto **BufferedWriter**, se necesita la clase **FileWriter**:

```
FileWriter fileW = new FileWriter(nombreFichero);
BufferedWriter bufferedW = new BufferedWriter(fileW);
```

3.2 FLUJOS DE BYTES

La clase abstracta **InputStream** representa las clases que producen entradas de distintas fuentes, como un array de bytes, un objeto **String**, un fichero, una tubería, etc. La clase abstracta **OutputStream** decide donde irá la salida, como a un array de bytes, un fichero o una tubería.

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de bytes.



Las clases **FileInputStream** y **FileOutputStream** manipulan los flujos de bytes que provienen o se dirigen hacia ficheros en disco.

3.2.1 Las clases FileInputStream y FileOutputStream

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que permiten trabajar con ficheros son **`FileInputStream`** (para entrada) y **`FileOutputStream`** (para salida), que operan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Cuando se instancia un objeto con la clase **`FileInputStream`**, el programa abre el fichero (que se debe enviar como argumento del constructor) en modo lectura. Una vez abierto, se podrá leer la información que contiene de forma secuencial byte a byte.

Constructores de la clase **`FileInputStream`**:

Constructor	Explicación	Excepción que lanza
`FileInputStream(String nombreDeFichero)`	Recibe como parámetro el nombre del fichero a abrir.	`FileNotFoundException` si el fichero no existe.
`FileInputStream(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar.	`FileNotFoundException` si el fichero no existe.

Principales métodos de la clase **`FileInputStream`**:

Método	Explicación
`int read()`	Devuelve el código ASCII del siguiente byte que hay después de donde está situado el puntero del fichero. Dicho puntero se va moviendo secuencialmente por el fichero, según vamos leyendo los bytes. Devuelve -1 si no hay ningún byte más que leer.
`int read(byte cadByte[])`	Lee hasta `cadByte.length` bytes guardándolos en la tabla que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.
`void close()`	Cierra el fichero.

```
import java.io.*;
public class Ejemplo03 {

	public static void main(String args[]) throws IOException{
		int c;
		try{
			//Se puede poner / o  \\
			FileInputStream f = new FileInputStream("/pedidos.txt");
			/*Al poner / va a buscar el fichero en la raíz del disco donde está
			  el proyecto. Si no se pone la /, busca el fichero en la carpeta
			  donde está ahora mismo*/
			//FileInputStream f = new FileInputStream("pedidos.txt");
			while((c=f.read())!=-1)
				System.out.print((char) c);
			/*Si no se hace la conversión visualizaría el código ASCII
			de cada carácter que hayguardado en el fichero*/
			f.close();
		}catch(FileNotFoundException e){
			System.out.println("El fichero no existe. ");
		}
	}
  }
```

Cuando se instancia un objeto con la clase **`FileOutputStream`**, lo que hace el programa es abrir el fichero para escritura. Una vez abierto, se podrá guardar información byte a byte. Si el fichero no existe, se crea en ese momento.

Constructores de la clase **`FileOutputStream`**:

Constructor	Explicación
`FileOutputStream(String nombreFich)`	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(File fichero)`	Recibe como parámetro un objeto `File` que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(String nombreDeFich, boolean append)`	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
`FileOutputStream(File fichero, boolean append)`	Recibe como parámetro un objeto `File` que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase **`FileOutputStream`**:

Método	Explicación
`int write(int byte)`	Escribe el byte que recibe como argumento en el fichero.
`int write(byte cadByte[])`	Escribe todos los bytes que contiene la tabla cadByte.
`void close()`	Cierra el fichero.

```
  import java.io.*;
public class Ejemplo04 {

      public static void main(String args[]) throws IOException{
            String cadena;
            int codCaracter;
            char caracter;

            FileOutputStream f1 = new FileOutputStream("fichejemplo04.txt ");
            cadena=Leer.pedirCadena("Introduce una frase: ");
            for(int pos=0;pos<cadena.length();pos++){
                  f1.write(cadena.charAt(pos));
            }
            f1.write('\n');//Para separar las frases
            f1.close();

            System.out.println(" \nEl contenido del fichero es: ");

            FileInputStream f2 = new FileInputStream("fichejemplo04.txt ");
            codCaracter=f2.read();
            while(codCaracter!=-1){
                  caracter=(char)codCaracter;
                  System.out.print(caracter);
                  codCaracter=f2.read();
            }
            f2.close();
      }
   }
```

3.2.2 Las clases DataInputStream y DataOutputStream

Para leer y escribir datos de tipos primitivos (como `boolean`, `int`, `long`, `float`, `double`, `char`, etc.), se utilizan las clases **`DataInputStream`** y **`DataOutputStream`**. Además de los métodos **`read`**() y **`write`**(), estas clases

proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina. Algunos de los métodos disponibles son:

Métodos para lectura	Métodos para escritura
boolean readBoolean()	void writeBoolean(boolean b)
byte readByte()	voidwriteByte(int i)
int readUnsignedByte()	void writeBytes(String s)
int readUnsignedShort()	void writeShort(int i)
short readShort()	void writeChars(String s)
char readChar()	void writeChar(int i)
int readInt()	void writeInt(int i)
long readLong()	void writeLong(long l)
float readFloat()	void writeFloat(float f)
double readDouble()	void writeDouble(double d)
String readUTF()	void writeUTF(String s)

Para abrir un objeto **DataInputStream**, se utilizan los mismos métodos que para **FileInputStream**. Por ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\FichData.dat");
FileInputStream fileIS = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(fileIS);
```

Para abrir un objeto **DataOutputStream**, se utilizan los mismos métodos que para **FileOutputStream**. Por ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\FichData.dat");
FileOutputStream fileOS = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileOS);
```

Hay que tener mucho cuidado con leer un fichero en el mismo formato que se ha escrito porque, de no ser así, daría errores en la ejecución al no corresponderse los tipos.

Para saber que se ha alcanzado el final del fichero, los métodos lanzan la excepción **EOFException**, así que hay que recogerla y tratarla correctamente. Ejemplo:

```
import java.io.*;
public class Ejemplo05 {

	public static void main(String args[]) throws FileNotFoundException{
		int numInt;
		String cadena;
		float numFloat;

		FileOutputStream fichEscrit = new FileOutputStream("prueba.dat");
		DataOutputStream escritTipos = new DataOutputStream(fichEscrit);

		FileInputStream fichLect = new FileInputStream("prueba.dat");
		DataInputStream lectTipos = new DataInputStream(fichLect);

		try {
			numInt=Leer.pedirEntero("Inserta un número entero:");
			escritTipos.writeInt(numInt);

			numFloat=Leer.pedirFloat("Inserta un número con decimales: ");
			escritTipos.writeFloat(numFloat);

			cadena=Leer.pedirCadena("Inserta una cadena:");
			escritTipos.writeChars(cadena);
			escritTipos.close();

			numInt=lectTipos.readInt();
			numFloat=lectTipos.readFloat();
			cadena=lectTipos.readLine();

			System.out.println(" \nLos datos leídos del fichero son: \n");
```

```
                System.out.println("-**:" +numInt+"-**:" +numFloat+"-**:" +cadena);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

3.2.3 Las clases ObjectInputStream y ObjectOutputStream

Se llama **persistencia** al proceso de almacenar toda la información que contiene un objeto, manteniendo su estructura, en un fichero binario. En Java, para poder guardar un objeto en un fichero binario, dicho objeto tiene que implementar la interfaz **`Serializable`**, que dispone de métodos que permiten escribir y leer objetos en ficheros binarios:

Método para escribir un objeto	**`void writeObject(ObjectOutputStream stream) throws IOException`**
Método para leer un objeto	**`void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`**

La **serialización** de objetos de Java permite tomar cualquier objeto que implemente la interfaz `Serializable` y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer y escribir objetos serializables a un stream se utilizan las clases **`ObjectInputStream`** y **`ObjectOutputStream`** respectivamente.

Por ejemplo, para escribir un objeto "persona" en un fichero binario, se necesita crear un flujo de salida a disco con FileOutputStream, y a continuación, crear el flujo de salida **ObjectOutputStream**, que procesa los datos y está vinculado al fichero. Por último, el método writeObject() escribe el objeto al flujo de salida y lo guarda en el fichero.

```
File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
FileOutputStream fileOS = new FileOutputStream(fichero);
ObjectOutputStream objectOS = new ObjectOutputStream(fileOS);
objectOS.writeObject(persona);
```

Por ejemplo, para leer un objeto "persona" de un fichero binario, se necesita crear el flujo de entrada a disco **`FileInputStream`**, y a continuación, crear el flujo de entrada **`ObjectInputStream`**, que procesa los datos y está vinculado al fichero. Por último, el método **`readObject`**() lee el objeto del flujo de entrada y puede lanzar la excepción **`ClassNotFoundException`**.

```
File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
FileInputStream fileIS = new FileInputStream(fichero);
ObjectInputStream objectIS = new ObjectInputStream(fileIS);
persona = (Persona) objectIS.readObject();
```

4 ACCESO ALEATORIO

Hasta ahora, todas las operaciones sobre ficheros se han realizado de forma secuencial. Java dispone de la clase **`RandomAccessFile`** para posicionar un cursor en una posición concreta de un fichero y acceder a su contenido de forma aleatoria o directa (no secuencial). Esta clase no es parte de la jerarquía **`Reader/Writer`** ni **`InputStream/OutputStream`**, puesto que permite avanzar y retroceder dentro de un fichero.

Constructores de la clase **`RandomAccessFile`**:

Método	Explicación
`RandomAccessFile(String nombre, String modo)`	nombre es el nombre del fichero y modo es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).
`RandomAccessFile(File fichero, String modo)`	fichero es el objeto fichero y modo es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).

La clase **`RandomAccessFile`** maneja un puntero o cursor que indica la posición actual en el fichero. Cuando se abre un fichero, el puntero se coloca en 0, es decir, apuntando al principio del mismo. Además, esta clase implementa las **interfaces `DataInput`** y **`DataOutput`**. Por tanto, una vez abierto un fichero, se pueden utilizar sus métodos **`readXXX`***()* y **`writeXXX`***()* para cada tipo de dato y las sucesivas llamadas a estos métodos **`read`***()* y **`write`***()* ajustan el puntero según la cantidad de bytes leídos o escritos.

Principales métodos de la clase **`RandomAccessFile`**:

Método	Explicación
`long getFilePointer()`	Devuelve la posición actual del puntero del fichero.
`void seek(longposicion)`	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo.
`long length()`	Devuelve el tamaño del fichero en bytes y marca el final del fichero.
`int skipBytes(int desplazamiento)`	Desplaza el puntero del fichero desde la posición actual el número de bytes indicados en desplazamiento.

Cada tipo de dato tiene un tamaño concreto:

`boolean` (1 bit)	**`byte`** (1 byte)	**carácter `Unicode`** (2 bytes)
`short` (2 bytes)	**`int`** (4 bytes)	**`long`** (8 bytes)
`float` (4 bytes)	**`double`** (8 bytes)	

El siguiente programa Java abre un fichero para lectura y escritura, escribe los nombres que se leen por teclado en el fichero al final, y después los lee del fichero para mostrarlos en pantalla.

```
import java.io.*;
public class Ejemplo06 {

	public static void main(String args[]) throws IOException{
		RandomAccessFile fichAleatorio;
		String nomNuevo,nombre;
		fichAleatorio = new RandomAccessFile("directo06.txt","rw");

		nomNuevo=Leer.pedirCadena("Introduce un nombre (\"*\" para salir):");
		while (!nomNuevo.equals("*")){
			fichAleatorio.seek(fichAleatorio.length());
			fichAleatorio.writeBytes(nomNuevo);
			fichAleatorio.write('\n');//Para separar los nombres
			nomNuevo=Leer.pedirCadena("Introduce nombre (\"*\" para salir):");
		}

		fichAleatorio.seek(0);
		nombre=fichAleatorio.readLine();
		while(nombre!=null) {
			System.out.println("Nombre leído: " +nombre);
			nombre=fichAleatorio.readLine();
		}
		fichAleatorio.close();
	}
 }
```

El siguiente ejemplo inserta datos de empleados (apellido, departamento y salario) en un fichero aleatorio. Estos datos se introducen de manera secuencial (no se usa el método *seek()*), y además, por cada empleado, se inserta un identificador que coincide con el índice+1 con el que se recorren los vectores. La longitud del registro de cada empleado es la misma (36 bytes).

```
import java.io.*;
public class EjemCreaEmpleAleatorio {

	public static void main(String[] args) throws IOException {
		File fichero = new File("AleatorioEmple.dat");
		//declara el fichero de acceso aleatorio
		RandomAccessFile file = new RandomAccessFile(fichero, "rw");
		//arrays con los datos
		String apellido[] = {"FERNANDEZZZZZ","GIL","LOPEZ","MARTINEZ",
				"SEVILLA","CEREZO", "REY"}; //apellidos
		int dep[] = {10, 20, 10, 10, 30, 30, 20};	//departamentos
		Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
				2200.0, 1435.87, 2000.0};//salarios

		StringBuffer buffer = null;//buffer para almacenar apellido
		int n=apellido.length;//numero de elementos del array

		for (int i=0;i<n; i++){ //recorro los arrays
			file.writeInt(i+1); //uso i+1 para identificar empleado
			buffer = new StringBuffer( apellido[i] );
			buffer.setLength(10); //10 caracteres para el apellido
			file.writeChars(buffer.toString());//insertar apellido
			file.writeInt(dep[i]);	//insertar departamento
			file.writeDouble(salario[i]);//insertar salario
		}
		file.close();  //cerrar fichero
	}

 }
```

El siguiente ejemplo usa el fichero anterior y visualiza todos sus registros. El posicionamiento para empezar a recorrer los registros empieza en 0, y para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento.

```
import java.io.*;
public class EjemLeeEmpleAleatorio {

	public static void main(String[] args) throws IOException {
		File fichero = new File("AleatorioEmple.dat");
		//declara el fichero de acceso aleatorio
		RandomAccessFile file = new RandomAccessFile(fichero, "r");
		//
		int  id, dep, posicion;
		Double salario;
		char apellido[] = new char[10], aux;

		posicion = 0;  //para situarnos al principio
		file.seek(posicion); //nos posicionamos en posicion

		//recorro el fichero
		while(file.getFilePointer() < file.length()){  //Si he recorrido todos los bytes
salgo del bucle
			id = file.readInt();   // obtengo id de empleado
			//recorro uno a uno los caracteres del apellido
			for (int i = 0; i < apellido.length; i++) {
				aux = file.readChar();
				apellido[i] = aux;    //los voy guardando en el array
			}

			//convierto a String el array
			String apellidos = new String(apellido);
			dep = file.readInt();        //obtengo dep
			salario = file.readDouble(); //obtengo salario

			System.out.printf("ID: %s, Apellido: %s, Departamento: %d, Salario: %.2f %n",
					id,   apellidos.trim(), dep, salario);

			//me posiciono para el sig empleado, cada empleado ocupa 36 bytes
			posicion= posicion + 36;
			file.seek(posicion); //nos posicionamos en posicion
		}//fin bucle while
		file.close();  //cerrar fichero
	}

}
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero. Conociendo su identificador, se puede acceder a la posición que ocupa dentro del fichero y obtener sus datos.