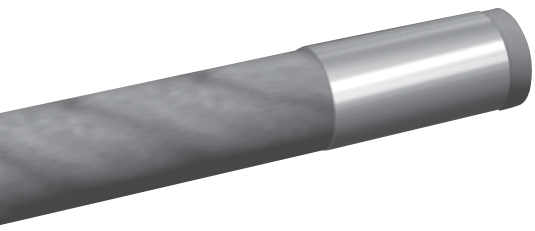
Capítulo 35

Archivos y flujos en Java



Contenido

- Flujos y archivos
- Clase `File`
- Flujos y jerarquía de clases
- Archivos de caracteres: flujos de tipo `Reader` y `Writer`
- Archivos de objetos
- Archivos de acceso directo
- Resumen
- Ejercicios
- Problemas

Introducción

Los archivos tienen como finalidad guardar datos de forma permanente; una vez que acaba la aplicación, los datos siguen disponibles para que otra aplicación pueda recuperarlos, para su consulta o modificación.

El proceso de archivos en Java se hace mediante el concepto de **flujo** (*stream*) o canal, o también denominado secuencia. Los *flujos* pueden estar abiertos o cerrados, conducen los datos entre el programa y los dispositivos externos. Con las clases y sus métodos proporcionados por el *paquete* `java.io` se pueden tratar archivos secuenciales, de acceso directo, archivos indexados, etcétera.

Conceptos clave

- Acceso secuencial
- Archivos de texto
- Flujos

Flujos y archivos

Un ***fichero*** (archivo) de datos, o simplemente un **archivo**, es una colección de registros relacionados entre sí con aspectos en común y organizados para un propósito específico. Por ejemplo, un fichero de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase.

Un archivo en una computadora es una estructura diseñada para contener datos, los cuales están organizados de tal modo que puedan ser recuperados fácilmente, actualizados o borrados y almacenados de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

- *acceso secuencial,*
- *acceso directo.*

El *acceso secuencial* implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.

El *acceso directo* implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionales.

En Java un archivo es, sencillamente, una secuencia de bytes, que son la representación de los datos almacenados. Java dispone de clases para trabajar las secuencias de bytes como datos de tipos básicos (`int`, `double`, `String` ...); incluso, para escribir o leer del archivo objetos. El diseño del archivo es el que establece la forma de manejar las secuencias de bytes, con una organización secuencial, o bien de acceso directo.

Un **flujo** (*stream*) es una abstracción que se refiere a una *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal (*pipe*) por la que circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene el archivo; por el canal que comunica el archivo con el programa van a fluir las secuencias de datos.

Abrir un archivo supone crear un objeto que queda asociado con un flujo. Al comenzar la ejecución de un programa Java se crean automáticamente tres objetos de flujo, son tres canales por los que pueden *fluir datos*, de entrada o de salida. Éstos son objetos definidos en la clase `System`:

- **`System.in;`** entrada estándar; permite la entrada al programa de flujos de bytes desde el teclado.
- **`System.out;`** salida estándar; permite al programa la salida de datos por pantalla.
- **`System.err;`** salida estándar de errores; permite al programa salida de errores por pantalla.

En Java, un archivo es simplemente un flujo externo, una secuencia de bytes almacenados en un dispositivo externo (normalmente en disco). Si el archivo se abre para salida, es un flujo de archivo de salida. Si el archivo se abre para entrada, es un flujo de archivo de entrada. Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo o a otro. El flujo es por tanto una abstracción de tal forma que las operaciones que realizan los programas son sobre el flujo independientemente del dispositivo al que esté asociado.

> ***A tener en cuenta***
> El paquete `java.io` agrupa al conjunto de clases e interfaces necesarios para procesar archivos. Es necesario utilizar clases de este paquete; por consiguiente, se debe incorporar al programa con la sentencia `import java.io.*`.

Clase `File`

Un archivo consta de un nombre, además de la ruta que indica dónde está ubicado, por ejemplo `"C:\pasaje.dat"`. Este identificador del archivo (cadena de caracteres) se transmite al constructor del flujo de entrada o de salida que procesa el fichero:

```
FileOutputStream f = new FileOutputStream("C:\pasaje.dat");
```

Los constructores de flujos están sobrecargados para, además de recibir el archivo como cadena, recibir un objeto de la clase `File`. Este tipo de objeto contiene el nombre del archivo, la ruta y más propiedades relativas al archivo.

La clase `File` define métodos para conocer propiedades del archivo (*última modificación, permisos de acceso, tamaño, ...*); también métodos para modificar alguna característica del archivo.

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.

`public File(String nombreCompleto)`
Crea un objeto File con el nombre y ruta del archivo pasado como argumento.

`public File(String ruta, String nombre)`
Crea un objeto File con la ruta y el nombre del archivo pasado como argumentos.

`public File(File ruta, String nombre)`
Crea un objeto File con un primer argumento que a su vez es un objeto File con la ruta y el nombre del archivo como segundo argumento.

Por ejemplo:

```
File miFichero = new File("C:\LIBRO\Almacen.dat");
```

crea un objeto `FILE` con el archivo `Almacen.dat` que está en la ruta `C\LIBRO`.

```
File otro = new File("COCINA", "Enseres.dat");
```

> ***Nota***
> Es una buena práctica crear objetos `File` con el archivo que se va a procesar, para pasar el objeto al constructor del flujo en vez de pasar directamente el nombre del archivo. De esta forma se pueden hacer controles previos sobre el archivo.

Información de un archivo

Con los métodos de la clase `File` se obtiene información relativa al archivo o ruta con que se ha inicializado el objeto. Así, antes de crear un flujo para leer de un archivo es conveniente determinar si el archivo existe; en caso contrario no se puede crear el flujo. A continuación se exponen los métodos más útiles para conocer los atributos de un archivo o un directorio.

`public boolean exists( )`
Devuelve true si existe el archivo (o el directorio).

`public boolean canWrite( )`
Devuelve true si se puede escribir en el archivo, si no es de sólo lectura.

`public boolean canRead( )`
Devuelve true si es de sólo lectura.

`public boolean isFile( )`
Devuelve true si es un archivo.

`public boolean isDirectory( )`
Devuelve true si el objeto representa a un directorio.

`public boolean isAbsolute( )`
Devuelve true si el directorio es la ruta completa.

`public long length( )`
Devuelve el número de bytes que ocupa el archivo. Si el objeto es un directorio devuelve cero.

`public long lastModified( )`
Devuelve la hora de la última modificación. El número devuelto es una representación interna de la hora, minutos y segundos de la última modificación, sólo es útil para establecer comparaciones con otros valores devueltos por el mismo método.

También dispone de métodos que modifican el archivo: *cambiar de nombre, marcar de sólo lectura*. Para los directorios, tiene métodos para crear uno nuevo, obtener una lista de todos los elementos (archivos o subdirectorios) del directorio. Algunos de estos métodos se escriben a continuación.

`public String getName( )`
Devuelve una cadena con el nombre del archivo o del directorio con que se ha inicializado el objeto.

`public String getPath( )`
Devuelve una cadena con la ruta relativa al directorio actual.

`public String getAbsolutePath( )`
Devuelve una cadena con la ruta completa del archivo o directorio.

`public boolean setReadOnly( )`
Marca al archivo para que no se pueda escribir, de sólo lectura.

`public boolean delete( )`
Elimina el archivo o directorio (debe estar vacío).

`public boolean renameTo(File nuevo)`
Cambia el nombre del archivo con que ha sido inicializado el objeto por el nombre que contiene el objeto pasado como argumento.

`public boolean mkdir( )`
Crea el directorio con el que se ha creado el objeto.

`public String[ ] list( )`
Devuelve un array de cadenas, cada una contiene un elemento (archivo o directorio) del directorio con el que se ha inicializado el objeto.

Ejemplo 35.1

Se muestra por pantalla cada uno de los archivos y subdirectorios de que consta un directorio que se transmite en la línea de órdenes.

Se crea un objeto `File` inicializado con el nombre del directorio o ruta procedente de la línea de órdenes. El programa comprueba que es un directorio y llamando al método `list( )` se obtiene un arreglo de cadenas con todos los elementos del directorio. Con un bucle, tantas iteraciones como longitud tiene el arreglo de cadenas, se escribe cada cadena en la pantalla.

```
import java.io.*;
class Directorio
{
  public static void main(String[ ] a)
  {
    File dir;
    String[ ] cd;
    // para la ejecución es necesario especificar el directorio
    if (a.length > 0)
    {
      dir = new File(a[0]);
      // debe ser un directorio
      if (dir.exists( ) && dir.isDirectory( ))
      {
```

```
      // se obtiene la lista de elementos
      cd = dir.list( );
      System.out.println("Elementos del directorio " + a[0]);
      for (int i = 0; i < cd.length; i++)
        System.out.println(cd[i]);
    }
    else
      System.out.println("Directorio vacío");
  }
  else
    System.out.println("No se ha especificado directorio ");
  }
}
```

Flujos y jerarquía de clases

En los programas hay que crear objetos *stream* y en muchas ocasiones hacer uso de los objetos `in, out` de la clase `System`. Los flujos de datos, de caracteres, de bytes se pueden clasificar en flujos de entrada y flujos de salida. En consonancia, Java declara dos clases (derivan directamente de la clase `Object`): `InputStream` y `OutputStream`. Ambas son clases abstractas que declaran métodos que deben redefinirse en sus clases derivadas. `InputStream` es la clase base de todas las clases definidas para flujos de entrada, y `OutputStream` es la clase base de todas las clases definidas para flujos de salida. La tabla 35.1 muestra las clases derivadas más importantes de éstas.

Archivos de *bajo nivel*: `FileInputStream` y `FileOutputStream`

Todo archivo, para entrada o salida, se puede considerar como una secuencia de bytes. A partir de estas secuencias de bytes, *flujos de bajo nivel*, se construyen flujos de más alto nivel para proceso de datos complejos, desde tipos básicos hasta objetos. Las clases `FileInputStream` y `FileOutputStream` se utilizan para leer o escribir bytes en un archivo; objetos de estas dos clases son los flujos de entrada y salida, respectivamente, a nivel de bytes. Los constructores de ambas clases permiten crear flujos (objetos) asociados a un archivo que se encuentra en cualquier dispositivo; el archivo queda abierto. Por ejemplo, el flujo `mf` se asocia al archivo `Temperatura.dat` del directorio en forma predeterminada:

```
FileOutputStream mf = new FileOutputStream("Temperatura.dat");
```

Las operaciones que a continuación se realicen con `mf` escriben secuencias de bytes en el archivo `Temperatura.dat`.

La clase `FileInputStream` dispone de métodos para leer un byte o una secuencia de bytes. A continuación se escriben los métodos más importantes de esta clase, todos con visibilidad `public`. Es importante tener en cuenta la excepción que pueden *lanzar* para que cuando se invoquen se haga un tratamiento de la excepción.

Tabla 35.1 Primer nivel de la jerarquía de clases de Entrada/Salida.

InputStream	OutputStream
FileInputStream	FileOutputStream
PipedInputStream	PipedOutputStream
ObjectInputStream	ObjectOutputStream
StringBufferInputStream	FilterOutputStream
FilterInputStream	

`FileInputStream(String nombre) throws FileNotFoundException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

`FileInputStream(File nombre) throws FileNotFoundException;`
Crea un objeto inicializado con el objeto archivo pasado como argumento.

`int read( ) throws IOException;`
Lee un byte del flujo asociado. Devuelve –1 si alcanza el fin del fichero.

`int read(byte[ ] s) throws IOException;`
Lee una secuencia de bytes del flujo y se almacena en el arreglo `s`. Devuelve –1 si alcanza el fin del fichero, o bien el número de bytes leídos.

`int read(byte[ ] s, int org, int len) throws IOException;`
Lee una secuencia de bytes del flujo y se almacena en arreglo `s` desde la posición `org` y un máximo de `len` bytes. Devuelve –1 si alcanza el fin del fichero, o bien el número de bytes leídos.

`void close( )throws IOException;`
Cierra el flujo, el archivo queda disponible para posterior uso.

La clase `FileOutputStream` dispone de métodos para escribir bytes en el flujo de salida asociado a un archivo. Los constructores inicializan objetos con el nombre del archivo, o bien con el archivo como un objeto `File`; el archivo queda abierto. A continuación se escriben los constructores y métodos más importantes; todos tienen visibilidad `public`.

`FileOutputStream(String nombre) throws IOException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

`FileOutputStream(String nombre, boolean sw) throws IOException;`
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento. En el caso de que `sw = true` los bytes escritos se añaden al final.

`FileOutputStream(File nombre) throws IOException;`
Crea un objeto inicializado con el objeto archivo pasado como argumento.

`void write(byte a) throws IOException;`
Escribe el byte `a` en el flujo asociado.

`void write(byte[ ] s) throws IOException;`
Escribe el arreglo de bytes en el flujo.

`void write(byte[ ] s, int org, int len) throws IOException;`
Escribe el arreglo `s` desde la posición `org` y un máximo de `len` bytes en el flujo.

`void close( )throws IOException;`
Cierra el flujo, el archivo queda disponible para posterior uso.

Nota
Una vez creado un flujo pueden realizarse operaciones típicas de archivos, *leer* (flujos de entrada), *escribir* (flujos de salida). Es el constructor el encargado de abrir el flujo; en definitiva, de abrir el archivo. Si el constructor no puede crear el flujo (archivo de lectura no existe…), lanza la excepción `FileNotFoundException`.

Recomendación
Siempre que finaliza la ejecución de un programa Java se cierran los flujos abiertos. Sin embargo, se aconseja ejecutar el método `close( )` cuando deje de utilizarse un flujo; de esa manera se liberan recursos asignados y el archivo queda disponible.

Ejemplo 35.2

Dado el archivo `jardines.txt`*, se desea escribir toda su información en el archivo* `jardinOld.txt`.

El primer archivo se asocia con un flujo de entrada; el segundo fichero, con un flujo de salida. Entonces, se instancia un objeto *flujo de entrada* y otro de salida del tipo `FileInputStream` y `FileOutputStream`, respectivamente. La lectura se realiza byte a byte con el método `read( );` cada byte se escribe en el flujo de salida invocando al método `write( )`. El proceso termina cuando `read( )` devuelve `–1`, señal de haber alcanzado el fin del archivo.

```
import java.io.*;
public class CopiaArchivo
{
  public static void main(String [ ] a)
  {
    FileInputStream origen = null;
    FileOutputStream destino = null;
    File f1 = new File("jardines.txt");
    File f2 = new File("jardinOld.txt");
    try
    {
      origen = new FileInputStream(f1);
      destino = new FileOutputStream(f2);
      int c;
      while ((c = origen.read( )) != -1)
        destino.write((byte)c);
    }
    catch (IOException er)
    {
      System.out.println("Excepción en los flujos "
                                      + er.getMessage( ));
    }
    finally {
      try
      {
        origen.close( );
        destino.close( );
      }
      catch (IOException er)
      {
        er.printStackTrace( );
      }
    }
  }
}
```

Archivos de datos: `DataInputStream` y `DataOutputStream`

Resulta poco práctico trabajar directamente con flujos de bytes; los datos que se escriben o se leen en los archivos son más elaborados, de más alto nivel, como enteros, reales, cadenas de caracteres, etc. Las clases `DataInputStream` y `DataOutputStream` (derivan de `FilterInputStream` y `FilterOutputStream`, respectivamente) *filtran* una secuencia de bytes, organizan los bytes para formar datos de tipo primitivo. De esta forma, se pueden escribir o leer directamente datos de tipo: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `String`.

La clase `DataInputStream` declara el comportamiento de los flujos de entrada, con métodos para leer cualquier tipo básico: `readInt( )`, `readDouble( )`, `readUTF( )`, etc. Estos flujos leen los bytes de otro flujo de bajo nivel (flujo de bytes) para formar números enteros, doble precisión, etc.; por consiguiente, deben asociarse a un flujo de bytes, de tipo `InputStream`, generalmente un

flujo `FileInputStream`. La asociación se realiza al crear el flujo, el constructor recibe como argumento el objeto flujo de *bajo nivel*, de bytes, del que realmente se lee. Por ejemplo, para leer datos del archivo `nube.dat`, se crea un flujo `gs` de tipo `FileInputStream` asociado con el archivo, a continuación un objeto `DataInputStream` que envuelve al flujo `gs`:

```
FileInputStream gs = new FileInputStream("nube.dat");
DataInputStream ent = new DataInputStream(gs);
```

Se puede escribir en una sola sentencia:

```
DataInputStream ent = new DataInputStream(
                      new FileInputStream("nube.dat"));
```

Al crear el flujo `ent` queda asociado con el flujo de bajo nivel `FileInputStream`, que a su vez abre, en modo lectura, el archivo `nube.dat`.

A continuación se escriben los métodos más importantes de la clase `DataInputStream`. Todos tienen visibilidad `public`, y además no se pueden redefinir, ya que están declarados como `final`.

`public DataInputStream(InputStream entrada) throws IOException`
Crea un objeto asociado con cualquier objeto de entrada pasado como argumento.

`public final boolean readBoolean( ) throws IOException`
Devuelve el valor de tipo `boolean` leído.

`public final byte readByte( ) throws IOException`
Devuelve el valor de tipo `byte` leído.

`public final short readShort( ) throws IOException`
Devuelve el valor de tipo `short` leído.

`public final char readChar( ) throws IOException`
Devuelve el valor de tipo `char` leído.

`public final int readInt( ) throws IOException`
Devuelve el valor de tipo `int` leído.

`public final long readLong( ) throws IOException`
Devuelve el valor de tipo `long` leído.

`public final float readFloat( ) throws IOException`
Devuelve el valor de tipo `float` leído.

`public final double readDouble( ) throws IOException`
Devuelve el valor de tipo `double` leído.

`public final String readUTF( ) throws IOException`
Devuelve una cadena que se escribió en formato UTF.

El flujo de entrada lee un archivo que previamente ha sido escrito con un flujo de salida del tipo `DataOutputStream`. Los flujos de salida se asocian con otro flujo de salida de bajo nivel, de bytes. Los métodos de este tipo de flujos permiten escribir cualquier valor de tipo de dato primitivo, `int double ...` y `String`. Al constructor de `DataOutputStream` se pasa como argumento el flujo de bajo nivel al cual queda asociado. De esta forma, el método, por ejemplo, `writeInt( )` que escribe un número entero, en realidad escribe 4 bytes en el flujo de salida.

La aplicación que vaya a leer un archivo creado con los métodos de un flujo `DataOutputStream` debe tener en cuenta que los métodos de lectura se han de corresponder. Por ejemplo, si se han escrito secuencias de datos de tipo *entero*, *char* y *double* con los métodos `writeInt( )`, `writeChar( )` y `writeDouble( )`, el flujo de entrada para leer el archivo utilizará los métodos `readInt( )`, `readChar( )` y `readDouble( )`, respectivamente.

El flujo de salida que se crea para escribir el archivo `nube.dat`:

```
FileOutputStream fn = new FileOutputStream("nube.dat");
DataOutputStream snb = new DataOutputStream(fn);
```

O bien, en una sola sentencia:

```
DataOutputStream snb = new DataOutputStream(
                       new FileOuputStream("nube.dat"));
```

A continuación se escriben los métodos más importantes de `DataOutputStream`:

`public DataOutputStream(OutputStream destino) throws IOException`
Crea un objeto asociado con cualquier objeto de salida pasado como argumento.

`public final void writeBoolean(boolean v) throws IOException`
Escribe el dato de tipo `boolean v`.

`public final void writeByte(int v) throws IOException`
Escribe el dato `v` como un byte.

`public final void writeShort(int v) throws IOException`
Escribe el dato `v` como un short.

`public final void writeChar(int v) throws IOException`
Escribe el dato `v` como un carácter.

`public final void writeChars(String v) throws IOException`
Escribe la secuencia de caracteres de la cadena `v`.

`public final void writeInt(int v) throws IOException`
Escribe el dato de tipo `int v`.

`public final void writeLong(long v) throws IOException`
Escribe el dato de tipo `long v`.

`public final void writeFloat(float v) throws IOException`
Escribe el dato de tipo `float v`.

`public final void writeDouble(double v) throws IOException`
Escribe el valor de tipo `double v`.

`public final void writeUTF(String cad) throws IOException`
Escribe la cadena `cad` en formato UTF. Escribe los caracteres de la cadena y dos bytes adicionales con longitud de la cadena.

`public final int close( )throws IOException`
Cierra el flujo de salida.

Nota

La composición de flujos es la forma habitual de *filtrar* secuencias de bytes para tratar datos a más *alto nivel*, desde datos de tipos básicos como `int`, `char`, `double` hasta objetos como pueden ser cadenas, arreglos y cualquier objeto creado por el usuario. Así, para leer un archivo datos a través de un *búfer*:

```
DataInputStream entrada = new DataInputStream(
                          new BufferedInputStream(
                          new FileInputStream(miFichero)));
```

Ejercicio 35.1

Se dispone de los datos registrados en la estación meteorológica situada en el cerro Garabitas, correspondientes a un día del mes de septiembre. La estructura de los datos: un primer registro con el día (p. ej., 1 septiembre); y para cada hora: hora, presión y temperatura. Los datos se han de grabar en el archivo `SeptGara.tmp`.

Para resolver el supuesto enunciado se define un flujo del tipo `DataOutputStream` asociado a otro flujo de salida a *bajo nivel* o simplemente flujo de bytes. Con el fin de simular la situación real, los datos se obtienen aleatoriamente, llamando al método `Math.random( )` y transformando el número aleatorio en otro dentro de un rango preestablecido. Se utilizan los métodos `writeUTF( )`, `writeDouble( )` y `writeInt( )` para escribir en el objeto flujo. En cuanto al tratamiento de excepciones, simplemente se captura y se escribe el mensaje asociado.

```
import java.io.*;
public class Garabitas
{
  public static void main(String[ ] a)
  {
    String dia = "1 Septiembre 2001";
    DataOutputStream obfl = null;
    try {
      obfl = new DataOutputStream (
                new FileOutputStream("septGara.tmp"));
      obfl.writeUTF(dia); // escribe registro inicial
      for (int hora = 0; hora < 24; hora++)
      {
        double presion, temp;
        presion = presHora( );
        temp = tempHora( );
        // escribe según la estructura de cada registro
        obfl.writeInt(hora);
        obfl.writeDouble(presion);
        obfl.writeDouble(temp);
      }
    }
    catch (IOException e)
    {
      System.out.println(" Anomalía en el flujo de salida " +
                         e.getMessage( ));
    }
    finally {
      try
      {
        obfl.close( );
      }
      catch (IOException er)
      {
        er.printStackTrace( );
      }
    }
  }
  // métodos auxiliares para generar temperatura y presión
  static private double presHora( )
  {
    final double PREINF = 680.0;
    final double PRESUP = 790.0;
    return (Math.random( )*(PRESUP - PREINF) + PREINF);
  }
  static private double tempHora( )
  {
    final double TEMINF = 5.0;
    final double TEMSUP = 40.0;
    return (Math.random( )*(TEMSUP - TEMINF) + TEMINF);
  }
}
```

Ejercicio 35.2

El archivo `SeptGara.tmp` ha sido creado con un flujo `DataOutputStream`, tiene la estructura de datos: el primer registro es una cadena escrita con formato UTF; los demás registros tienen los datos: hora (tipo `int`), presión (tipo `double`) y temperatura (tipo `double`). Se quiere escribir un programa para leer cada uno de los datos, calcular la temperatura máxima y mínima y mostrar los resultados por pantalla.

El archivo que se va a leer se creó con un flujo `DataOutputStream`; entonces para leer los datos del archivo se necesita un flujo `DataInputStream` y conocer la estructura, en cuanto a tipos de datos, de los ítems escritos. Como esto se conoce, se procede a la lectura: primero se lee la cadena inicial con la fecha (`readUTF( )`); a continuación se leen los campos correspondientes a la hora, temperatura y presión con los métodos `readInt( )`, `readDouble( )` y `readDouble( )`. Hay que leer todo el archivo mediante el típico bucle *mientras no fin de fichero*; sin embargo, la clase `DataInputStream` no dispone del método `eof( )`. Por ello el bucle está diseñado como *un bucle infinito*; la salida del bucle se produce cuando un método intente leer después de *fin de fichero*, entonces lanza la excepción `EOFException` y termina el bucle.

Los datos leídos se muestran en pantalla; con llamadas al método `Math.max( )` se calcula el valor máximo pedido.

```
import java.io.*;
class LeeGarabitas
{
  public static void main(String[ ] a)
  {
    String dia;
    double mxt = -11.0; // valor mínimo para encontrar máximo
    FileInputStream f;
    DataInputStream obfl = null;
    try {
      f = new FileInputStream("septGara.tmp");
      obfl = new DataInputStream(f);
    }
    catch (IOException io)
    {
       System.out.println("Anomalía al crear flujo de entrada, " +
                          io.getMessage( ));
       return; // termina la ejecución
    }
    // proceso del flujo
    try {
      int hora;
      boolean mas = true;
      double p, temp;
      dia = obfl.readUTF( );
      System.out.println(dia);
      while (mas)
      {
        hora = obfl.readInt( );
        p = obfl.readDouble( );
        temp = obfl.readDouble( );
        System.out.println("Hora: " + hora + "\t Presión: " + p
                           + "\t Temperatura: " + temp);
        mxt = Math.max(mxt,temp);
      }
    }
    catch (EOFException eof)
    {
      System.out.println("Fin de lectura del archivo.\n");
    }
```

```
    catch (IOException io)
    {
      System.out.println("Anomalía al leer flujo de entrada, "
                         + io.getMessage( ));
      return; // termina la ejecución
    }
    finally {
      try
      {
        obfl.close( );
      }
      catch (IOException er)
      {
        er.printStackTrace( );
      }
    }
    // termina el proceso, escribe la temperatura máxima
    System.out.println("\n La temperatura máxima: " + (float)mxt);
  }
}
```

Flujos `PrintStream`

La clase `PrintStream` deriva directamente de `FilterOutputStream`. La característica más importante es que dispone de métodos que añaden la marca de fin de línea. Los flujos de tipo `PrintStream` son de salida, se asocian con otro flujo de bajo nivel, de bytes, que a su vez se crea asociado a un archivo externo. Por ejemplo, mediante el flujo `fjp` se puede escribir cualquier tipo de dato; los bytes que ocupa cada dato se vuelcan secuencialmente en `Complex.dat`

```
fjp = new PrintStream(new FileOutputStream("Complex.dat"));
```

Los métodos de esta clase, `print( )` y `println( )` están sobrecargados para poder escribir desde cadenas hasta cualquiera de los datos primitivos; `println( )` escribe un dato y a continuación añade la marca de fin de línea.

A continuación están los métodos más importantes:

`public PrintStream(OutputStream destino)`
: Crea un objeto asociado con cualquier objeto de salida pasado como argumento.

`public PrintStream(OutputStream destino, boolean flag)`
: Crea un objeto asociado con objeto de salida pasado como argumento y si el segundo argumento es `true` se produce un automático volcado al escribir el fin de línea.

`public void flush( )`
: Vuelca el flujo actual.

`public void print(Object obj)`
: Escribe la representación del objeto `obj` en el flujo.

`public void print(String cad)`
: Escribe la cadena en el flujo.

`public void print(char c)`
: Escribe el carácter c en el flujo.

`método print( ) para cada tipo de dato primitivo.`

`public void println(Object obj)`
: Escribe la representación del objeto `obj` en el flujo y el fin de línea.

`public void println(String cad)`
: Escribe la cadena en el flujo y el fin de línea.

`public void println(char c)`

Escribe el carácter `c` en el flujo y el fin de línea.

`método println( ) para cada tipo de dato primitivo`

Nota de programación

El objeto definido en la clase `System: System.out` es de tipo `PrintStream`, asociado, normalmente, con la pantalla. Por ello se han utilizado los métodos:

```
System.out.print( );
System.out.println( );
```

Archivos de caracteres: flujos de tipo `Reader` y `Writer`

Los flujos de tipo `InputStream` y `OutputStream` están orientados a bytes, para el trabajo con flujos orientados a caracteres Java dispone de las clases de tipo `Reader` y `Writer`.

`Reader` es la clase base, abstracta, de la jerarquía de clases para leer un carácter o una secuencia de caracteres. `Writer` es la clase base de la jerarquía de clases diseñadas para escribir caracteres.

Leer archivos de caracteres: `InputStreamReader`, `FileReader` y `BufferedReader`

Para leer archivos de caracteres se utilizan flujos derivados de la clase `Reader`. Ésta declara métodos para lectura de caracteres que son heredados, en algún caso redefinido, por las clases derivadas. Los métodos más interesantes:

`public int read( )`

Lee un carácter; devuelve el carácter leído (como un entero). Devuelve –1 si lee el final del archivo.

`public int read(char [ ] b);`

Lee una secuencia de caracteres, hasta completar el array b, o bien leer el carácter fin de archivo. Devuelve el número de caracteres leídos, o bien –1 si alcanzó el final del archivo.

`InputStreamReader`

Los flujos de la clase `InputStreamReader` *envuelven* a un flujo de bytes; convierte la secuencia de bytes en secuencia de caracteres y de esa forma lee caracteres en lugar de bytes. La clase deriva directamente de la clase `Reader`, por lo que tiene disponibles los métodos `read( )` para lectura de caracteres. Generalmente se utilizan estos flujos como entrada en la construcción de flujos con búfer.

El método más importante es el constructor que tiene como argumento cualquier flujo de entrada:

```
public InputStreamReader(InputStream ent);
```

En el siguiente ejemplo se crea el flujo `entradaChar` que puede leer caracteres del flujo de bytes `System.in` (asociado con el teclado):

```
InputStremReader entradaChar = new  InputStreamReader(System.in);
```

`FileReader`

Para leer *archivos de texto*, archivos de caracteres, se puede crear un flujo del tipo `FileReader`. Esta clase deriva de `InputStreamReader`, hereda los métodos `read( )` para lectura de caracteres; el constructor tiene como entrada una cadena con el nombre del archivo.

```
public FileReader(String miFichero) throws FileNotFoundException;
```

Por ejemplo,

```
FileReader fr = new FileReader("C:\cartas.dat");
```

No resulta eficiente, en general, leer directamente de un flujo de este tipo; se utilizará un flujo `BufferedReader` *envolviendo* al flujo `FileReader`.

BufferedReader

La lectura de archivos de texto se realiza con un flujo que almacena los caracteres en un búfer intermedio. Los caracteres no se leen directamente del archivo sino del búfer; con esto se consigue más eficiencia en las operaciones de entrada. La clase `BufferReader` permite crear flujos de caracteres con búfer, es una forma de organizar el flujo básico de caracteres del que procede el texto; esto se manifiesta en que al crear el flujo `BufferReader` se inicializa con un flujo de caracteres de tipo `InputStreamReader`.

El constructor de la clase tiene un argumento de tipo `Reader`, un `FileReader` o un `InputStreamReader`. El flujo creado dispone de un búfer de un tamaño, normalmente suficiente; el tamaño del búfer se puede especificar en el constructor con un segundo argumento aunque no resulta necesario. Ejemplos de flujos con búfer:

```
File mf = new File("C:\listados.txt");
FileReader fr = new FileReader(mf);
BufferedReader bra = new BufferedReader(fr);

File mfz = new File("Complejo.dat");
BufferedReader brz = new BufferedReader(
                        new InputStreamReader(
                          new FileInputStream(mfz)));
```

La clase `BufferedReader` deriva directamente de la clase `Reader`. Entonces, dispone de los métodos `read( )` para leer un carácter, o bien un arreglo de caracteres. El método más importante es `readLine( )`:

```
public String readLine( ) throws IOException;
```

El método lee una línea de caracteres, termina con el carácter de fin de línea, y devuelve una cadena con la línea leída (no incluye el carácter fin de línea). Puede devolver `null` si lee la marca de fin de archivo.

Otro método interesante es `close( )`, cierra el flujo y libera los recursos asignados. El fin de la aplicación Java también cierra los flujos abiertos, aunque es recomendable cerrar un flujo que no se va a utilizar.

```
public void close( ) throws IOException;
```

Ejemplo 35.3

Para ver el contenido de un archivo de texto se van a leer línea a línea hasta leer la marca de fin de fichero. El nombre completo del archivo se ha de transmitir en la línea de órdenes de la aplicación.

Para realizar la lectura del archivo de texto, se va a crear un flujo `BufferReader` asociado con el flujo de caracteres del archivo. Se crea un objeto `File` con argumento, la cadena transmitida en la *línea de órdenes*. Una vez creado el flujo, el bucle *mientras "cadena leída distinto de null"* procesa todo el archivo.

```
import java.io.*;
class LeerTexto
{
  public static void main(String[ ] a)
  {
    File mf;
    BufferedReader br = null;
```

```
  String cd;
  // se comprueba que hay una cadena
  if (a.length > 0)
  {
    mf = new File(a[0]);
    if (mf.exists( ))
    {
      int k = 0;
      try {
        br = new BufferedReader(new FileReader(mf));
        while ((cd = br.readLine( )) != null)
        {
          System.out.println(cd);
          if ((++k)%21 == 0)
          {
            System.out.print("Pulse una tecla ...");
            System.in.read( );
          }
        }
        br.close( );
      }
      catch (IOException e)
      {
        System.out.println(e.getMessage( ));
      }
    }
    else
      System.out.println("Directorio vacío");
    }
   }
}
```

Flujos que escriben caracteres: `Writer`, `PrintWriter`

Los archivos de texto son archivos de caracteres; se pueden crear con flujos de bytes, o bien con flujos de caracteres derivados de la clase abstracta `Writer`.

La clase `Writer` define métodos `write( )` para escribir arreglos de caracteres o cadenas (`String`). De `Writer` deriva `OutputStreamWriter` que permite escribir caracteres en un flujo de bytes al cual se asocia en la creación del objeto o flujo. Por ejemplo:

```
OutputStreamWriter ot = new OutputStreamWriter(
                        new FileOutputStream(archivo));
```

No es frecuente utilizar directamente flujos `OutputStreamWriter`. Ahora bien, resulta de interés porque la clase `FileWriter` es una extensión de ella, diseñada para escribir en un archivo de caracteres. Los flujos de tipo `FileWriter` escriben caracteres, método `write( )`, en el archivo al que se asocia el flujo cuando se crea el objeto.

```
FileWriter nr = new FileWriter("cartas.dat");
nr.write("Estimado compañero de fatigas");
```

`PrintWriter`

Los flujos más utilizados para salida de caracteres son de tipo `PrintWriter`. Esta clase declara constructores para asociar un flujo `PrintWriter` con cualquier otro flujo de tipo `Writer`, o bien `OutputStream`.

`public PrintWriter(OutputStream destino)`

Crea un flujo asociado con cualquier flujo de salida a nivel de byte.

`public PrintWriter(Writer destino)`

Crea un flujo asociado con otro flujo de salida de caracteres de tipo Writer.

La importancia de esta clase radica en que define los métodos `print( )` y `println( )` para cada uno de los tipos de datos simples, para `String` y para `Object`. La diferencia entre los métodos `print( )` y `println( )` está en que `println( )` añade, a continuación de los caracteres escritos para el argumento, los caracteres de fin de línea.

`public void print(Object obj)`
Escribe la representación del objeto `obj` en el flujo.

`public void print(String cad)`
Escribe la cadena en el flujo.

`public void print(char c)`
Escribe el carácter c en el flujo.
Sobrecarga de `print( )` para cada tipo de dato primitivo.

`public void println(Object obj)`
Escribe la representación del objeto `obj` en el flujo y el fin de línea.

`public void println(String cad)`
Escribe la cadena en el flujo y el fin de línea.

`public void println(char c)`
Escribe el carácter `c` en el flujo y el fin de línea.
Sobrecarga de `println( )` para cada tipo de dato primitivo.

Ejemplo 35.4

En un archivo de texto se van a escribir los caminos directos entre los pueblos de una comarca alcarreña. Cada línea contiene el nombre de dos pueblos y la distancia del camino que los une (sólo si hay camino directo) separados por un blanco. La entrada de los datos es a partir del teclado.

El archivo de texto, argumento de la línea de órdenes, se maneja con un objeto `File` de tal forma que si ya existe se crea un flujo de bytes `FileOutputStream` con la opción de añadir al final. Este flujo se utiliza para componer un flujo de más alto nivel, `DataOutputStream`, que a su vez se le asocia con el flujo `PrintWriter` para escribir datos con los métodos `print` y `println`. La entrada es a partir del teclado; se repiten las entradas hasta que el método `readLine( )` devuelve `null` (cadena vacía) debido a que el usuario ha tecleado `^Z`; o bien termina al pulsar una línea vacía. La cadena leída inicializa un objeto de la clase `StringTokenizer` para comprobar que se han introducido correctamente los datos pedidos.

```
import java.io.*;
import java.util.StringTokenizer;
class EscribeCamino
{
  static boolean datosValidos(String cad) throws Exception
  {
    StringTokenizer cd;
    String dist;
    boolean sw;
    cd = new StringTokenizer(cad);
    sw = cd.countTokens( ) == 3;
    cd.nextToken( );
    sw = sw && (Integer.parseInt(cd.nextToken( )) > 0);
    return sw;
  }
  public static void main(String[ ] a)
  {
    File mf;
    BufferedReader entrada = new BufferedReader(
                             new InputStreamReader(System.in));
    DataOutputStream d = null;
```

```
  PrintWriter pw = null;
  String cad;
  boolean modo;
  // se comprueba que hay una cadena
  if (a.length > 0)
  {
    mf = new File(a[0]);
    if (mf.exists( ))
      modo = true;
    else
      modo = false;
    try {
      pw = new PrintWriter( new DataOutputStream (
             new FileOutputStream(mf,modo)));
      System.out.println("Pueblo_A distancia Pueblo_B");
      while (((cad = entrada.readLine( ))!= null)&&
              (cad.length( ) > 0))
      {
        if (datosValidos(cad))
          pw.println(cad);
      }
      pw.close( );
    }
    catch (Exception e)
    {
      System.out.println(e.getMessage( ));
      e.printStackTrace( );
    }
  }
  else
      System.out.println("Archivo no existente ");
  }
}
```

Archivos de objetos

Para que un objeto persista una vez que termina la ejecución de una aplicación se ha de guardar en un archivo de objetos. Por cada objeto que se almacena en un archivo se graban características de la clase y los atributos del objeto. Las clases `ObjectInputStream` y `ObjectOutputStream` están diseñadas para crear flujos de entrada y salida de objetos persistentes.

Clase de objeto *persistente*

La declaración de la clase cuyos objetos van a persistir debe implementar la interfaz `Serializable` del paquete `java.io`. Es una interfaz vacía, no declara métodos; simplemente indica a la JVM que las instancias de estas clases podrán grabarse en un archivo. La clase declarada a continuación tiene esta propiedad:

```
import java.io.*;
class Racional implements Serializable {...}
```

Nota de programación

Los atributos de los objetos declarados `transient` no se escriben al grabar un objeto en un archivo.

```
class TipoObjeto implements Serializable
{
  transient tipo dato; // no se escribe en el flujo de objetos
}
```

Flujos ObjectOutputStream

Los flujos de la clase ObjectOutputStream se utilizan para grabar objetos persistentes. El método writeObject( ) escribe cualquier objeto de una clase *serializable* en el flujo de bytes asociado; puede elevar excepciones del tipo IOException que es necesario procesar.

```
public void writeObject(Object obj) throws IOException;
```

El constructor de la clase espera un argumento de tipo OutputStream, que es la clase base de los flujos de salida a nivel de bytes. Entonces, para crear este tipo de flujos, primero se crea un flujo de salida a nivel de bytes (asociado a un archivo externo) y a continuación se pasa este flujo como argumento al constructor de ObjectOutputStream. Por ejemplo:

```
FileOutputStream bn = new FileOutputStream("datosRac.dat");
ObjectOutputStream fobj = new ObjectOutputStream(bn);
```

O bien en una sola sentencia:

```
ObjectOutputStream fobj = new ObjectOutputStream(
                              new FileOutputStream("datosRac.dat"));
```

A continuación se puede escribir cualquier tipo de objeto en el flujo:

```
Racional rd = new Racional(1,7);
fobj.writeObject(rd);
String sr = new String("Cadena de favores");
fpbj.writeObj(sr);
```

Flujos ObjectInputStream

Los objetos guardados en archivos con flujos de la clase ObjectOutputStream se recuperan, se leen, con flujos de entrada del tipo ObjectInputStream. Esta clase es una extensión de InputStream, además implementa la interfaz DataInput; por ello dispone de los diversos métodos de entrada (read) para cada uno de los tipos de datos, readInt( )… El método más interesante definido por la clase ObjectInputStream es readObject( ); lee un objeto del flujo de entrada (del archivo asociado al flujo de bajo nivel). Este objeto se escribió en su momento con el método writeObject( ).

```
public Object readObject( ) throws IOException;
```

El constructor de flujos ObjectInputStream tiene como entrada otro flujo, de bajo nivel, de cualquier tipo derivado de InputStream, por ejemplo FileInputStream, asociado con el archivo de objetos. A continuación se crea un flujo de entrada para leer los objetos del archivo "archivoObjets.dat":

```
ObjectInputStream obje = new ObjectInputStream(
                         new FileInputStream("archivoObjets.dat "));
```

El constructor levanta una excepción si, por ejemplo, el archivo no existe,..., del tipo ClassNotFoundException, o bien IOException; es necesario poder capturar estas excepciones.

> ***Nota***
> El método readObject( ) lee cualquier objeto del flujo de entrada, devuelve el objeto como tipo Object. Entonces, es necesario convertir Object al tipo del objeto que se espera leer. Por ejemplo, si el archivo es de objetos Racional:
>
> ```
> rac = (Racional) flujo.readObject( );
> ```
>
> La lectura de archivos con diversos tipos de objetos necesita una estructura de selección para conocer el tipo de objeto leído. El operador instanceof es útil para esta selección.

Ejercicio 35.3

Se desea guardar en un archivo los libros y discos de que disponemos. Los datos de interés para un libro son: título, autor, editorial, número de páginas; para un disco: cantante, título, duración en minutos, número de canciones y precio.

Se podría diseñar una jerarquía de clases que modelase los objetos libro, disco,… Sin embargo, el ejercicio sólo pretende mostrar cómo crear objetos persistentes; declara directamente la clase `Libro` y `Disco` con la propiedad de ser "serializables" (implementan la interfaz `java.io.Serializable`). La clase principal crea un flujo de salida para objetos; según los datos que introduce el usuario se instancia un tipo de objeto `Disco` o `Libro` y con el método `writeObject( )` se escribe en el flujo.

```
import java.io.*;
class Libro implements Serializable
{
  private String titulo;
  private String autor;
  private String editorial;
  private int pagina;
  public Libro( )
  {
    titulo = autor = editorial = null;
  }
  public Libro(String t, String a, String e, int pg)
  {
    titulo = t;
    autor = a;
    editorial = e;
    pagina = pg;
  }
  public void entrada(BufferedReader ent) throws IOException
  {
    System.out.print("Titulo: "); titulo = ent.readLine( );
    System.out.print("Autor: "); autor = ent.readLine( );
    System.out.print("Editorial: "); editorial = ent.readLine( );
    System.out.print("Páginas: ");
    pagina = Integer.parseInt(ent.readLine( ));
  }
}
class Disco implements Serializable
{
  private String artista;
  private String titulo;
  private int numCancion, duracion;
  private transient double precio;
  public Disco( )
  {
    artista = titulo = null;
  }
  public Disco(String a, String t, int n, int d, double p)
  {
    titulo = t;
    artista = a;
    numCancion = n;
    duracion = d;
    precio = p;
  }
  public void entrada(BufferedReader ent) throws IOException
  {
    System.out.print("Cantante: "); artista = ent.readLine( );
    System.out.print("Titulo: "); titulo = ent.readLine( );
```

```
    System.out.print("Canciones: ");
    numCancion = Integer.parseInt(ent.readLine( ));
    System.out.print("Duración(minutos): ");
    duracion = Integer.parseInt(ent.readLine( ));
    System.out.print("Precio: ");
    precio = Double.valueOf(ent.readLine( )).doubleValue( );
  }
}
public class Libreria
{
  public static void main (String [ ] a)
  {
    BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in));
    File mf = new File("libreria.dat");
    ObjectOutputStream fobj = null;
    Libro libro = new Libro( );
    Disco disco = new Disco( );
    int tipo;
    boolean mas = true;
    try {
      fobj = new ObjectOutputStream(new FileOutputStream(mf));
      do {
        System.out.println
                ("Pulsa L(libro), D(disco), F(finalizar)");
        tipo = System.in.read( );
        System.in.skip(2); // salta caracteres fin de línea
        switch (tipo) {
          case 'L':
          case 'l': libro = new Libro( );
                    libro.entrada(br);
                    fobj.writeObject(libro);
                    break;
          case 'D':
          case 'd': disco = new Disco( );
                    disco.entrada(br);
                    fobj.writeObject(disco);
                    break;
          case 'F':
          case 'f': fobj.close( );
                    mas = false;
        }
      } while (mas);
    }
    catch (IOException e)
    {
      e.printStackTrace( );
    }
  }
}
```

Archivos de acceso directo

Un archivo es de acceso aleatorio o directo cuando cualquier registro es directamente accesible mediante la especificación de un *índice*, que da la posición del registro con respecto al origen del archivo. La principal característica de los archivos de acceso aleatorio radica en la rapidez de acceso a un determinado registro; conociendo el *índice* del registro se puede situar el *puntero* de acceso en la posición donde comienza el registro. Las operaciones que se realizan con los archivos de acceso directo son las usuales: creación, consulta, dar altas, dar bajas, modificar.

Java considera un archivo como secuencias de bytes. A la hora de realizar una aplicación es cuando se establece la forma de acceso al archivo. Es importante, para el proceso de archivos aleatorios, establecer el tipo de datos de cada campo del registro lógico y el tamaño de cada registro. En cuanto al tamaño, se establece teniendo en cuenta la máxima longitud de cada campo; para los campos de tipo `String` se debe prever el máximo número de caracteres. Además, si se escribe en formato `UTF` se añaden 2 bytes más (en los dos bytes se guarda la longitud de la cadena). Los campos de tipo primitivo tienen longitud fija: `char` 2 bytes, `int` 4 bytes, `double` 8 bytes …

RandomAccessFile

La clase `RandomAccesFile` define métodos para facilitar el proceso de archivos de acceso directo. Esta clase deriva directamente de la clase `Object` e implementa los métodos de las interfaces `DataInput` y `DataOutput`, al igual que las clases `DataInputStream` y `DataOutputStream`, respectivamente. Por consiguiente, `RandomAccesFile` tiene tanto métodos de lectura como de escritura, que coinciden con los de las clases `DataInputStream` y `DataOutputStream`, además de métodos específicos para el tratamiento de este tipo de archivos.

Constructor

`public RandomAccessFile(String nomArchivo, String modo)`

El objeto queda ligado al archivo que se pasa como cadena en el primer argumento. El segundo argumento es el modo de apertura.

`public RandomAccesFile(File archivo, String modo)`

El primer argumento es un objeto `File` que se creó con la ruta y el nombre del archivo, el segundo argumento es el modo de apertura.

Java simplifica el modo de apertura al máximo; puede ser de dos formas:

`"r"`	Modo de sólo lectura. Únicamente se pueden realizar operaciones de entrada, de lectura de registros. No se pueden modificar o escribir registros.
`"rw"`	Modo lectura/escritura. Permite hacer operaciones tanto de entrada (lectura) como de salida (escritura).

Por ejemplo, los flujos a crear para acceder al archivo `Libros.dat` son:

```
File f = new File("libros.dat");
try
{
  RandomAccessFile dlib = new RandomAccessFile(f, "r");
}
catch (IOException e)
{
  System.out.println("Flujo no creado, " +
                     "el proceso no puede continuar");
  System.exit(1);
}
```

Ambos constructores lanzan una excepción del tipo `IOException` si hay algún problema al crear el flujo. Por ejemplo, si se abre para lectura y el archivo no existe. Es habitual comprobar el atributo de *existencia* del archivo con el método `exists( )` de la clase `File` antes de empezar el proceso. Por ejemplo:

```
try
  {
    RandomAccessFile dlib = new RandomAccessFile("Provins.dat","rw");
  }
  catch (IOException e)
  {
    System.out.println("Flujo no creado, " +
                       "el proceso no puede continuar");
    System.exit(1);
  }
```

Métodos de posicionamiento

En los archivos cuando se hace mención al *puntero* del archivo se refiere a la posición (en número de bytes) a partir de la que se va a realizar la siguiente operación de lectura o de escritura. Una vez realizada la operación, el *puntero* queda situado justo después del último byte leído o escrito. Por ejemplo, si el puntero se encuentra en la posición `n` y se lee un campo entero (`4` bytes) y un campo `double` (`8` bytes), la posición del *puntero* del archivo será `n+12`.

getFilePointer( )

Este método de la clase `RandomAccessFile` devuelve la posición, en número de bytes, actual del *puntero del archivo*. Su declaración:

```
public long getFilePointer( ) throws IOException
```

Por ejemplo, en la siguiente llamada al método, éste devuelve cero debido a que la posición inmediatamente después de abrir el flujo es cero.

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

seek ( )

Este método desplaza el puntero del archivo `n` bytes, tomando como origen el del archivo (byte `0`). Su declaración:

```
public void seek(long n) throws IOException
```

Con el método `seek( )` se puede tratar un archivo como un array que es una estructura de datos de acceso aleatorio. `seek( )` sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento que se pasa como argumento.

> ***Precaución***
> El desplazamiento pasado al método `seek( )` es relativo a la posición `0` del archivo. Es erróneo pasar desplazamientos negativos. Si el desplazamiento pasado es mayor que el tamaño actual del archivo, se amplía éste si a continuación se realiza una escritura.

length( )

Con este método se obtiene el tamaño actual del archivo, el número de bytes que ocupa o longitud que tiene. La declaración:

```
public long length( ) throws IOException
```

Por ejemplo, para conocer la longitud del archivo *Atletas*:

```
RandomAccessFile acc = new RandomAccessFile("Atletas.dat","rw");
System.out.println("Tamaño del archivo" + acc.length( ));
```

Para situar el puntero al final del archivo:

```
acc.seek(acc.length( ));
```

Proceso que crea y añade registros en un archivo directo

Para crear un archivo directo, lo primero a determinar es la longitud que va a tener cada registro. Esto depende de los campos y del número de bytes que tenga cada uno. Por ejemplo, suponer que se crea un archivo de acceso directo o aleatorio, con estas características:

Se dispone de una muestra de las coordenadas de puntos en el plano representada por pares de números reales `(x,y)`*, tales que* `1.0≤ x ≤100.0` *y* `1.0≤ y≤100.0`*. Cada punto es la ubicación*

en el plano de una casa o refugio rural identificado por su nombre popular ("Casa Mariano", "Refugio De los Ríos" ...). El nombre más largo es de 40 caracteres. Además, cada Casa y el punto en el plano está identificado por un número de orden. Esta muestra se va a guardar en un archivo con acceso directo, de tal forma que cada registro represente un punto en el plano con los campos: coordenadas del punto (x,y), nombre de la casa y número de orden. A su vez, el número de orden va a ser el número de registro.

El primer campo a considerar es `numOrden`, de tipo `int`, que ocupa `4` bytes; el campo `nombreCasa` de 40 caracteres como máximo, se va a escribir en formato `UTF` que añade `2` bytes más, por lo que ocupa un máximo de `82` bytes. Las coordenadas son dos números reales de tipo `double`, el tamaño es `8+8` bytes. En total cada registro tiene una longitud de `4+82+8+8= 102`.

El archivo va a tener el nombre de `Rurales.dat`. Se ha de crear un objeto `File` con el nombre del archivo y la ruta. El flujo (objeto `RandomAccessFile`) se abre en modo lectura/escritura, `"rw"` para así poder escribir cada registro.

El número de orden de cada registro se establece en el rango de `1` a `99`. Al comenzar la aplicación se crean los flujos y se llama al método `archVacio( )` para escribir `99` registros vacíos, con el campo `numOrden` a `-1`, el nombre de la casa a *cadena vacía* y las coordenadas `(0,0)`. Posiblemente queden *huecos*, o registros vacíos; posteriormente se pueden añadir nuevos registros.

La clase `CasaReg` tiene como variables instancia o atributos los campos de cada registro. El método `annadeReg( )` escribe en el flujo los campos del registro llamando a `writeInt( )`, `writeUTF( )` y `writeDouble( )`, respectivamente.

La entrada de los registros va a ser interactiva; la condición para acabar el bucle de entrada es leer como `NumOrden` el `0`.

```
import java.io.*;
import java.util.*;
class CasaReg
{
  int numOrden;
  String nombreCasa;
  double x, y;
  final int TAM = 102;
  public CasaReg( )
  {
    nombreCasa = new String( );
    numOrden = 0;
    x = y = 0.0;
  }
  public boolean annadeReg( RandomAccessFile fl)
  {
    boolean flag ;
    flag = true;
    try
    {
      fl.writeInt(numOrden);
      fl.writeUTF(nombreCasa);
      fl.writeDouble(x);
      fl.writeDouble(y);
    }
    catch (IOException e)
    {
      System.out.println("Excepción al escribir el registro.");
      flag = false;
    }
    return flag;
  }
  public int tamanno( ) { return TAM;}
  // desplazamiento en bytes de registro n
  public long desplazamiento(int n) { return TAM*(n-1);}
}
// clase principal
class CreaCasas
```

```
{
  File af;
  RandomAccessFile arch;
  CasaReg reg;
  final int MAXREG = 99;
  public CreaCasas( )
  {
    try
    {
      af = new File("Rurales.dat");
      arch = new RandomAccessFile(af, "rw");
      reg = new CasaReg( );
    }
    catch(IOException e)
    {
      System.out.println("No se puede abrir el flujo."
                         + e.getMessage( ));
      System.exit(1);
    }
  }
  void archVacio( )
  {
    try
    {
      reg = new CasaReg( ); // crea registro vacío
      for (int k = 1; k <= MAXREG; ++k)
      {
        arch.seek(reg.desplazamiento(k));
       reg.annadeReg(arch);
      }
    }
    catch(IOException e)
    {
      System.out.println("Excepción al mover puntero del archivo");
      System.exit(1);
    }
  }
  void procesoEntrada( )
  {
    Scanner entrada = new Scanner(System.in);
    System.out.println("Datos de la muestra en el orden: " +
          " Número de orden. Nombre. Coordenadas\n" +
          " Para terminar, Número de orden = 0");
    try {
      int nr;
      double x, y;
      String cd;
      boolean masRegistros = true;
      // bucle para la entrada
      while (masRegistros)
      {
        do {
          System.out.println(" Número de orden: ");
          nr = entrada.nextInt( );
      } while (nr<0 || nr>MAXREG);
      if (nr > 0) // no es clave de salida
      {
        reg.numOrden = nr;
        System.out.println(" Nombre: ");
        cd = entrada.nextLine( );
        // se asegura un máximo de 40 caracteres
        reg.nombreCasa = cd.substring(0,Math.min(40,cd.length( )));
        do {
            System.out.println(" Coordenadas (x,y): ");
            x = entrada.nextDouble( );
            y = entrada.nextDouble( );
```

```
            } while ((x<1.0 || x > 100.0) || (y<1.0 || y > 100.0));
            // escribe el registro en la posición que le corresponde
            reg.x = x;
            reg.y = y;
            arch.seek(reg.desplazamiento(nr));
            reg.annadeReg(arch);
          }
          masRegistros =( nr > 0);
        }   // fin del bucle de entrada
      }
      catch(IOException e)
      {
        System.out.println("Excepción, termina el proceso ");
        System.exit(1);
      }
      finally
      {
        System.out.println("Se cierra el flujo ");
        try {
          arch.close( );
        }
        catch(IOException e)
        {
          System.out.println("Error al cerrar el flujo ");
        }
      }
    }
    // driver del proceso
    public static void main(String [ ]a)
    {
      CreaCasas pr = new CreaCasas( );
      pr.archVacio( );
      pr.procesoEntrada( );
    }
  }
```

Consultas de registros en un archivo directo

En el apartado anterior, se ha creado un archivo aleatorio con las ubicaciones de casas en el plano con ciertas condiciones. Ahora se quiere realizar el proceso para hacer consultas y modificaciones. Normalmente hay diversos tipos de consultas, algunas de ellas:

- Listado de todos los registros.
- Mostrar un registro.
- Mostrar registros que cumplen cierta condición.

Para listar todos los registros hay que hacer una lectura de todo el archivo. Es un proceso completamente secuencial. Sin embargo, no todos los registros son altas realizadas, o registros activos. Así, si en nuestro archivo creado se ha previsto un máximo de `99` registros, no todos estarán dados de alta y por tanto habrá huecos. Los registros no activos son los que tienen como `numOrden 0`. Ésa será la clave para discernir si el registro leído se muestra. La lectura del archivo se hace hasta el final del archivo.

La segunda forma de consulta, por el número de registro, es más simple. Directamente se sitúa el puntero del archivo en la posición que determina el número de registro y a continuación se lee. El registro pedido puede ser que no esté activo, `numOrden = 0`. Si es así no se muestra.

La tercera forma de consulta supone el mismo proceso que la consulta total. Hay que leer todos los registros del archivo y seleccionar los que cumplen la condición.

Codificación

La clase `CasaReg`, además de tener un método para escribir los campos en un flujo, debe tener un método para leer los campos del registro de un flujo, por lo que se añade dicho método a la clase (los métodos anteriores no se vuelven a escribir).

Ahora se crea una nueva clase, `ConsultaCasa`, en la que se definen los métodos de consulta de todo el archivo o de un registro. El constructor recibe como argumento el objeto `File` con el archivo que se va a leer, abre el flujo en modo de *sólo lectura*. La clase principal crea el objeto `ConsultaCasa` y ofrece al usuario los diversos tipos de consulta.

```
import java.io.*;
import java.util.*;
class CasaReg
{
  int numOrden;
  String nombreCasa;
  double x, y;
  final int TAM = 102;
  // métodos ya escritos
  public CasaReg( ) {}
  public boolean annadeReg( ){}
  public int tamanno( ) { return TAM;}
  public long desplazamiento(int n) { return TAM*(n-1);}
  //
  public boolean leerReg( RandomAccessFile flo) throws EOFException
  {
    boolean flag ;
    flag = true;
    try
    {
      numOrden = flo.readInt( );
      nombreCasa = flo.readUTF( );
      x = flo.readDouble( );
      y = flo.readDouble( );
    }
    catch (EOFException e)
    {
      throw e;
    }
    catch (IOException e)
    {
      System.out.println("Excepción al leer el registro.");
      flag = false;
    }
    return flag;
  }
}
  // clase para realizar consultas
class ConsultaCasa
{
  private RandomAccessFile archivo;
  private void mostrar(CasaReg reg)
  {
    System.out.println(reg.nombreCasa +
                    " ubicada en las coordenadas \t (" +
                    (float)reg.x + "," + (float)reg.y +")");
  }
  public ConsultaCasa(File f) throws IOException
  {
    if (f.exists( ))
    {
      archivo = new RandomAccessFile(f,"r");
      System.out.println("Flujo abierto en modo lectura, tamaño: "
                          + archivo.length( ));
    }
    else
    {
      archivo = null;
      throw new IOException("Archivo no existe. ");
  }
}
```

```
public void consultaGeneral( ) throws IOException
{
  if (archivo != null)
  {
    try {
      int n;
      CasaReg reg = new CasaReg( );
      n = 0;
      // se lee hasta fin de archivo, termina con la
      // excepcion EOFException
      while (true)
      {
        archivo.seek(reg.desplazamiento(++n));
        if (reg.leerReg(archivo)) // lectura sin error
          if (reg.numOrden > 0) mostrar(reg); // registro activo
      }
    }
    catch(EOFException eof)
    {
      System.out.println("\n Se alcanza el fin del archivo" +
                         " en la consulta ");
    }
  }
}
public void consultaReg(int nreg) throws IOException
{
  if (archivo != null)
  {
    CasaReg reg = new CasaReg( );
    if (nreg>0 && reg.desplazamiento(nreg)< archivo.length( ))
    {
      archivo.seek(reg.desplazamiento(nreg));
      if (reg.leerReg(archivo))
        if (reg.numOrden > 0)
          mostrar(reg);
        else
          System.out.println("Registro " + nreg +
                             " no dado de alta.");
      }
      else throw new IOException(" Registro fuera de rango.");
    }
  }
}
//
// clase principal
public class Consultas
{
  public static void main(String [ ]a)
  {
    File f = new File("Rurales.dat");
    ConsultaCasa clta = null;
    int opc = 1;
    Scanner entrada = new Scanner(System.in);
    try {
      System.out.println("\n Consultas al archivo " +
                 f.getName( ));
      clta = new ConsultaCasa(f);
      while (opc != 5)
      {
        System.out.println("1. Muestra todos los registros " +
                                              "activos.");
        System.out.println("2. Muestra un registro determinado.");
        System.out.println("5. Se sale de la aplicación.");

        do {
```

```
            opc = entrada.nextInt( );
          } while ((opc<1) || (opc>2 && opc!=5));
          if (opc == 1)
            clta.consultaGeneral( );
          else if (opc == 2)
          {
            int nreg;
            System.out.print("Número de registro: ");
            nreg = entrada.nextInt( );
            clta.consultaReg (nreg);
          }
        }
      }
      catch(IOException e)
      {
        System.out.println("\n Excepción durante " +
              "el proceso de consulta: " +
              e.getMessage( ) + " Termina la aplicación.");
      }
    }
  }
```

Resumen

- Java dispone de un paquete especializado en la entrada y salida de datos, es el paquete `java.io`. Contiene un conjunto de clases organizadas jerárquicamente para tratar cualquier tipo de flujo de entrada o de salida de datos.
- Es necesaria la sentencia `import java.io.*` en los programas que utilicen objetos de alguna de estas clases.
- Todo se basa en la *abstracción* de flujo: *corriente* de bytes que entran a un dispositivo o que salen de él. Para Java un archivo, cualquier archivo, es un flujo de bytes; incorpora clases que procesan los flujos a bajo nivel, como secuencias de bytes.
- Para estructurar los bytes y formar datos a más alto nivel hay clases de más alto nivel; con estas clases se pueden escribir o leer directamente datos de cualquier tipo simple (entero, char...). Estas clases se enlazan con las clases de bajo nivel que a su vez se asocian a los archivos.
- Los objetos creados en Java pueden ser persistentes. Las clases de objetos persistentes implementan la interfaz `Serializable`. Los flujos que escriben y leen estos objetos son `ObjectOutputStream` y `ObjectInputStream`, respectivamente.
- Java dispone de la clase `RandomAccessFile` para procesar un archivo con acceso aleatorio. La clase `RandomAccessFile` contiene métodos de posicionamiento (`seek( )`), de lectura (`readInt( ), ...`) y de grabación de campos (`writeInt( ),...`).

Ejercicios

35.1. Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

35.2. Un archivo contiene enteros positivos y negativos. Escribir un método para leer el archivo y determinar el número de enteros negativos.

35.3. Escribir un método para copiar un archivo. El método tendrá dos argumentos de tipo cadena, el primero es el archivo original y el segundo el archivo destino. Utilizar flujos `FileInputStream` y `FileOutputStream`.

35.4. Una aplicación instancia objetos de las clases `NumeroComplejo` y `NumeroRacional`. La primera tiene dos variables instancia de tipo `float`, `parteReal` y `parteImaginaria`. La segunda clase tiene definidas tres variables, `numerador` y `denominador` de tipo `int` y `frac` de tipo `double`. Escribir la aplicación de tal forma que los objetos sean persistentes.

Problemas

35.1. Escribir un programa que compare dos archivos de texto (caracteres). El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.

35.2. Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones.

35.31 Las pruebas de acceso a la Universidad Pontificia de Guadalajara, UPGA, constan de 4 apartados, cada uno de los cuales se puntúa de 1 a 25 puntos. Escribir un programa para almacenar en un archivo los resultados de las pruebas realizadas, de tal forma que se escriban objetos con los siguientes datos: nombre del alumno, puntuaciones de cada apartado y la puntuación total.

35.4. Una farmacia quiere mantener su `stock` de medicamentos en un archivo. De cada producto interesa guardar el código, precio y descripción. Escribir un programa que genere el archivo pedido, almacenándose los objetos de manera secuencial.

35.5. Se quiere crear un archivo binario formado por registros que representan productos de perfumería. Los campos de cada registro son código de producto, descripción, precio y número de unidades. La dirección de cada registro viene dada por una función *hash* que toma como campo clave el código del producto (tres dígitos):

```
hash(clave) = (clave modulo 97) + 1
```

El número máximo de productos distintos es `100`. Las colisiones, de producirse, se situarán secuencialmente a partir del registro número `120`.