Unidad IV

Flujos y Archivos

Año 2008

> “Si nos soltaran al azar dentro del Cosmos la probabilidad de que nos encontráramos sobre un planeta o cerca de él sería inferior a una parte entre mil billones de billones (1 / 10 elevado a 33). En la vida diaria una probabilidad así se considera nula. Los mundos son algo precioso.”
>
> Kart Sagan, (Cosmos)

Autor: Ing. Tymoschuk, Jorge

Colaboración: Ing. Fritelli, Valerio

Unidad IV – Flujos y Archivos

Flujos y Archivos

En esta unidad veremos los métodos para la manipulación de archivos y directorios, así como los que se emplean para leer de, o escribir en, archivos. También estudiaremos el mecanismo de serialización de objetos mediante el cual podrá almacenar estos elementos de forma tan sencilla como si estuviera trabajando con datos de texto o numéricos.

Flujos

En esta unidad tratamos acerca de la forma de obtener información desde cualquier origen de datos capaz de enviar una secuencia de bytes y de cómo enviar información a cualquier destino que pueda también recibir una secuencia de datos. Estos orígenes y destinos de secuencias de bytes normalmente son **archivos**, aunque también podemos hablar de conexiones de red e, incluso, de bloques de memoria. Es interesante tener siempre en mente esta generalidad: por ejemplo, la información almacenada en archivos y la recuperada desde una conexión de red se manipula esencialmente de la misma forma. Desde luego, aunque los datos estén almacenados en último extremo como una secuencia de bytes, es mejor pensar en ellos como en una estructura de más alto nivel, como ser una secuencia de caracteres u objetos.

En Java, un objeto del cual podemos leer una secuencia de bytes recibe el nombre de flujo de entrada (o input stream), mientras que aquel en el que podemos escribir es un flujo de salida (u output stream). Ambos están especificados en las **clases abstractas InputStream** y **OutputStream**. Ya que los flujos orientados a byte **no** son adecuados para el procesamiento de información Unicode (recuerde que Unicode usa dos bytes para cada carácter), existe una jerarquía de clases separada para el procesamiento de estos caracteres que hereda de las **clases abstractas Reader** y **Writer**. Estas clases disponen de operaciones de lectura y de escritura basadas en los caracteres Unicode de 2 bytes, en lugar de caracteres de un solo byte.

Recuerde que el objetivo de una **clase abstracta** es ofrecer un mecanismo para agrupar el comportamiento común de clases a un nivel más alto. Esto lleva a aclarar el código y hacer el árbol de herencia más fácil de entender. La misma organización se sigue con la entrada y la salida en el lenguaje Java.

Java deriva estas cuatro clases abstractas en un gran numero de clases concretas. Veremos la mas usuales.

Un **flujo** *(stream)* es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen y un destino. 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 al archivo; por el canal que comunica el archivo con el programa fluyen las secuencias de datos. Abrir un archivo supone crear un objeto quedando asociado con un flujo.

Al comenzar la ejecución de un programa Java se crea automáticamente un objeto de la clase java.lang.**System**. Esta clase incluye estaticamente tres atributos que son **objetos flujo** de las clases abajo indicadas.

static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

System.err: Objeto para salida estándar de errores por pantalla.
System.in: Objeto para entrada estándar de datos desde el teclado.
System.out: Objeto para salida estándar de datos por pantalla.

Estos tres objetos sirven para procesar secuencias de caracteres en modo texto. Así, cuando se ejecuta System. out. print ("Aquí me pongo a cantar"); se escribe la secuencia de caracteres en pantalla, y cuando se ejecuta System. in. read () se capta un carácter desde teclado.

Todo esto lo hemos visto y usado extensamente.

Si el archivo se abre para salida, usaremos un flujo de salida. Si el archivo se abre para entrada, necesitamos un flujo 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.

El paquete **java.io** agrupa el conjunto de clases para el manejo de entrada y salida con archivos:
Siempre que se vaya a procesar un archivo se tienen que utilizar clases de este paquete, por lo que se debe de importar: import java. io.* si queremos el conjunto de todas las clases.

Todo el proceso de entrada y salida en lava se hace a través de flujos (stream). Los flujos de datos, de caracteres, de bytes se pueden clasificar en flujos de entrada (**Input Stream**) y en flujos de salida (**OutputStream**). Por ello Java declara dos clases abstractas que declaran métodos que deben redefinirse en sus clases derivadas. InputStream es la clase base de todas las clases definidas para streams de entrada, y OutputStream es la clase base de todas las clases de stream de salida.

Extienden **abstract InputStream**: FilelnputStream, ByteArraylnputStream, PipelnputStream, SequencelnputStream, StringBufferlnputStream, FilterlnputStream

Extienden **abstract OutputStream**: FileOutputStream, ByteArrayOutputStream, PipeOutputStream, FilterOutputStream

Clase FileInputStream

La clase FilelnputStream se utiliza para leer bytes desde un archivo. Proporciona operaciones básicas para leer un byte o una secuencia de bytes.

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

FilelnputStream(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 archivo.*

int read(byte[] s) throws IOException¡
Lee una secuencia de bytes del flujo y se almacena en el array s. *Devuelve* -1 *si alcanza el fin del*
archivo, 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 el array s *desde la posición org y un máximo de len bytes. Devuelve* -1 *si alcanza elfin del archivo. o bien el número de bytes leídos.*

Clase FileOutputStream

Con la clase FileOutputStream se pueden escribir bytes en un flujo de salida asociado a un archivo.

FileOutputStream(String nombre) throws IOException¡
Crea un objeto inicial izado 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 array de bytes en el flujo.*

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

Procesamiento Basico

Ejemplos usando FileInputStream y FileOutputStream

```
// La aplicación crea un objeto stream de salida, el archivo se denomina cadena.txt.
// El programa graba varios arrays de bytes inicializados desde cadenas,
// Para tratar las excepciones se define un bloque try y un catch de captura.
import java.io.*;
class ArchivoCad{
        private int contLin = 0;
        public void demo(){
                String str = new String();
                String lineas[] =       {"La mas antigua de todas las filosofias",
                                         "la de la evolucion",
                                         "estuvo maniatada de manos y pies",
                                         "y relegada a la oscuridad mas absoluta ..."};
                byte [] s;
                try {
```

```
Grabadas 4 lineas (exito)
Process Exit...
```

```
                FileOutputStream f = new FileOutputStream("cadena.txt");
                for(int i = 0; i < lineas.length;i++){
                        s = lineas[i].getBytes(); // copia la cadena en el array de bytes s;
                        f.write(s);               // graba el array de bytes
                        f.write((byte)'\n') ;     // graba el avance de carro
                        contLin++;                // cuenta
                }
        }
        catch (IOException e){System.out.println("Problema grabacion");}
        finally {System.out.println("Grabadas "+contLin+" lineas (exito)");}
    }


    public static void main(String [] args){
        ArchivoCad arch = new ArchivoCad();
        arch.demo();
    }
}
```

Nota: Si el archivo ya existía, es recubierto por el nuevo, sin aviso ni excepcion de ninguna clase

```
//  Al archivo ya creado queremos agregarle unas líneas, al final.
//  Todo lo que necesitamos es un segundo parámetro en  new FileOutputStream("cadena.txt", true);
import java.io.*;
class ArchivoCad02{
    private int contLin = 0;
    public void demo(){
        String str = new String();
        String lineas[] =  {"durante el milenio del escolasticismo teologico.",
                "Pero Darwin infundio nueva savia vital en la antigua estructura;",
                "las ataduras saltaron, y el pensamiento revivificado",
                "de la antigua Grecia ha demostrado ser una expresion mas",
                "adecuada del orden universal ... T.H.HUXLEY, 1887"};
        byte [] s;
        try {
                FileOutputStream f = new FileOutputStream("cadena.txt",true);
                for(int i = 0; i < lineas.length;i++){
                        s = lineas[i].getBytes(); // copia la cadena en el array de bytes s;
                        f.write(s);                                 // graba el array de bytes
                        f.write((byte)'\n') ;     // graba el avance de carro
                        contLin++;                                  // cuenta
                }
        }
        catch (IOException e){System.out.println("Problema grabacion");}
        finally {System.out.println("Agregadas "+contLin+" lineas (exito)");}
    }
    public static void main(String [] args){
        ArchivoCad02 arch = new ArchivoCad02();
        arch.demo();
    }
}
```

```
Agregadas 5 lineas (exito)
Process Exit...
```

```
//  queremos leer

import java.io.*;
class ArchivoLee {
   public void demo()
```

```
La mas antigua de todas las filosofias
la de la evolucion
estuvo maniatada de manos y pies
y relegada a la oscuridad mas absoluta ...
durante el milenio del escolasticismo teologico.
Pero Darwin infundio nueva savia vital en la antigua
estructura;
las ataduras saltaron, y el pensamiento revivificado
de la antigua Grecia ha demostrado ser una expresion mas
```

```
      int c;
      try {
         FileInputStream f = new FileInputStream("cadena.txt");
         while ((c = f.read()) !=-1)
             System.out.print((char)c) ;
      }
      catch(IOException e) {
          System.out.println("Anomalía flujo de entrada "+
            "(Revisar si el archivo existe) .");}
      finally {System.out.println("Lectura terminada !!!");}
   }
   public static void main(String [] args) {
      ArchivoLee lectura = new ArchivoLee();
      lectura.demo();
   }
}
```

Hemos visto ejemplos concretos de tratamiento de archivos de texto usando las clases **FileOutputStream** y **FileInputStream.** En Java las posibilidades de tratamiento de archivos son muy variadas. En nuestra asignatura pretendemos ver con algun detalle el tratamiento de archivos secuenciales y de acceso directo, por lo que solo citaremos a título informativo estas otras alternativas.

Clases ByteArrayInputS tream y ByteArrayOutputStream

Estas clases permiten asociar un flujo con un array de bytes, en vez de un archivo. Usandolas, podemos hacer que un objeto stream de entrada lea del array de bytes, y que un objeto stream de salida escriba en un array de bytes interno que crece dinámicamente.

Clases PipeInputStream y PipeOutputStream

Estas clases se utilizan para transferir datos entre tareas *(threads)* sincronizadas. Permite a dos tareas comunicarse mediante llamadas a los métodos de escritura y lectura. Se ha de definir un objeto stream de tipo *PipeInput* y otro objeto flujo de tipo *PipeOutput.* Para enviar datos a una tarea, el objeto flujo de salida invoca a la operación wri te ( ). La tarea que recibe datos los captura a través del objeto flujo de entrada, llamando a métodos de lectura, read() y receive( ). Ambas clases tienen un constructor al que se le pasa como argumento el objeto *pipe* de la otra clase; también tienen el método connect(), que tiene como argumento el objeto *pipe* con el que se conectan.

Clases Filtro

Los elementos transferidos, de entrada o de salida, con los flujos utilizados anteriormente han sido siempre secuencias de bytes. Los flujos *filtro* también leen secuencias de bytes, pero organizan internamente estas secuencias para formar datos de los tipos primitivos (int, long, double. . .) . Los stream *filtro* son una abstracción de las secuencias de bytes para hacer procesos de datos a más alto nivel; con esta abstracción ya no tratamos los items como secuencias o «*chorros*» de bytes, sino de forma elaborada con más funcionalidad. Así, a nivel lógico, se pueden tratar los datos dentro de un buffer, escribir o leer datos de tipo int, long, double directamente y no mediante secuencias de bytes. Los objetos stream *filtro* leen de un flujo que previamente ha tenido que ser escrito por otro objeto stream *filtro* de salida.

Considerando de forma aislada la jerarquía de clase stream *filtro* hay dos clases base: **FilterlnputStream** y **FilterOutputStream** (derivan directamente de InputSream y OutputStream). Son clases abstractas y por consiguiente la interfaz de cada clase tiene que ser definida en las clases derivadas.

La clase **FilterlnputStream** es extendida por BufferedlnputStream, LineNumberlnputStream, PushbacklnputStream y DataInputStream.

La clase **FilterOutputStream** es extendida por BufferedOutputStream, DataOutputStream y PrintStream.

Clases DataInputStream y DataOutputStream

La clase para entrada, DatalnputStream, *filtra* una secuencia de bytes, los organiza, para poder realizar lecturas de tipos de datos primitivos directamente: char, byte, short, int, long, float, double, boolean. Un objeto de esta clase lee de un flujo de entrada de bajo nivel (flujo de bytes) al que está asociado. La asociación se realiza en dos pasos:

```
1) - FileInputStream gs = new FileInputStream("Palabras.dat");
      //  Generamos objeto FileInputStream
2) - DataInputStream ent = new DataInputStream(gs);
      //  Generamos objeto DataInputStream
```

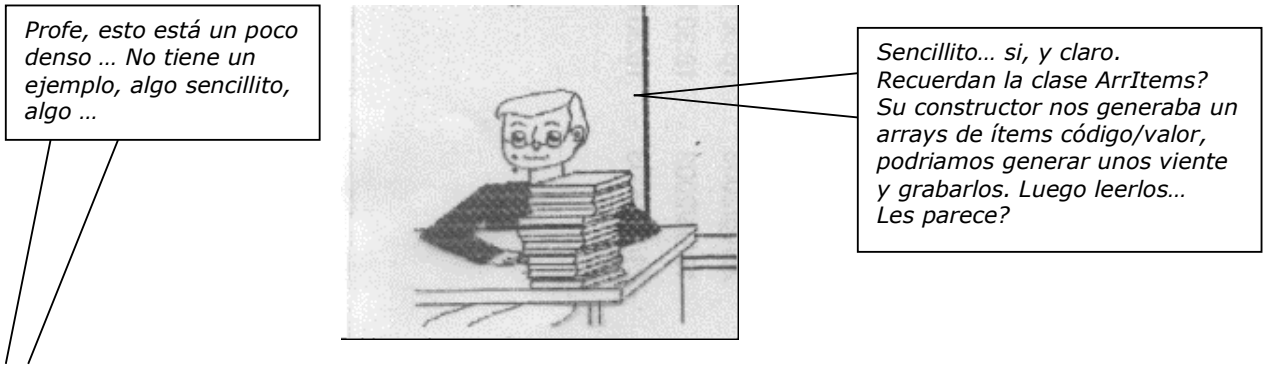
Algunos de los métodos más importantes de **DataInputStream**, todos son public y final.

```
DataInputStream(InputStream entrada)
// Constructor, requiere de parámetro tipo FileInputStream .

boolean readBoolean{) throws IOException
// Devuelve el valor de tipo boolean leído.

byte readByte() throws IOException  // Devuelve el valor de tipo byte leído.

short readShort() throws IOException // Devuelve el valor de tipo short leído.

char readChar{) throws IOException       // Devuelve el valor de tipo char leído.

int readlnt() throws IOException         // Devuelve el valor de tipo int leído.

long readLong() throws IOException       // Devuelve el valor de tipo long leído.

float readFloat() throws IOException     // Devuelve el valor de tipo float leído.

double readDouble() throws IOException   // Devuelve el valor de tipo double leído.

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

String readLine() throws IOException // Devuelve la cadena leída hasta fin de línea.
```

Algunos de los métodos más importantes de **DataOutputStream,** todos son public y final.

```
DataOutputStream(OutputStream destino) // Constructor, requiere de parámetro tipo FileOutputStream
void writeBoolean(boolean v) throws IOException // Escribe v de tipo boolean

void writeByte(int v) throws IOException  // Escribe v como un byte.

void writeShort(int v) throws IOException // Escribe v como un short.
void writeChar(int v) throws IOException  // Escribe v como un carácter.
```

```
void writeInt(int v) throws IOException  // Escribe  int v.
void writeLong(long v) throws IOException // Escribe el dato de tipo long v.
void writeFloat(float v) throws IOException // Escribe el dato de tipo float v.
void writeDouble(double v) throws IOException  // Escribe el valor de tipo double
v.
void writeUTF(String cad) throws IOException   //  Escribe  cadena  cad  en  formato
UTF.
int size()                                     // Devuelve el tamaño del flujo.
```



Grabando veinte elementos de Arritems

```
import ArrItems;
class GrabItems{
        private ArrItems aItems;
        private int itGrb;
        public GrabItems(int tam, char tipo){
                aItems = new ArrItems(tam,tipo);
                itGrb = 0;
        }

        public void demo()throws IOException{
                FileOutputStream f = new FileOutputStream("Archivo Items.tmp");
                DataOutputStream strm = new DataOutputStream(f) ;
                for(int i = 0; i <  aItems.talle;i++){
                        strm.writeInt(aItems.item[i].getCodigo());
                        strm.writeFloat(aItems.item[i].getValor());
                        itGrb++;
                }
                System.out.println("Grabados "+itGrb+" items");
        }

        public static void main(String[] args) throws IOException {
                GrabItems grbIt = new GrabItems(20,'R');
                grbIt.demo();
        }
}
```

```
Primeros 10 de 20
 elementos Item
1 – 16.220732
2 – 3.0816941
19 – 18.192905
6 – 9.128724
17 – 12.868467
11 – 3.3735917
1 – 7.5119925
4 – 7.7511096
4 – 14.572884
3 – 10.45162
```

Leyendo "Archivo Ítems.tmp"

```
import java.io.*;
class LeerItems{
        private int itLei;
        public LeerItems(){
                itLei = 0;
        }
        public String toString(){
```

```
Valores leidos
Archivo Items.tmp
1 – 16.220732
2 – 3.0816941
19 – 18.192905
6 – 9.128724
17 – 12.868467
. . . . .
Otros 12 items
. . . . .
9 – 6.0928297
8 – 16.284492
1 – 1.2296109
```

```
            FileInputStream f;
            DataInputStream strm;
            int cod;
            float val;
            String aux = "Valores leidos \nArchivo Items.tmp\n";
            try{
                  f = new FileInputStream("Archivo Items.tmp");
                  strm = new DataInputStream(f);
                  while (true){
                        cod = strm.readInt();
                        val = strm.readFloat();
                        aux+= cod+" - "+val+"\n";
                        itLei++;
                  }
                        }
            catch(EOFException eof){aux+="lectura terminada \n";}
            catch(IOException io){aux+="Anomalia al procesar flujo";}
            finally{aux+="Leidos "+itLei+" items \n";}
            return aux;
      }
      public static void main(String[] args) throws IOException {
            LeerItems leeIt = new LeerItems();
            System.out.println(leeIt);
      }
}
```

Nota: Observese que fin de archivo de entrada es tratado como excepción. De ello se ocupa **EOFException** (End Of File Exception)

La clase File

En todos los ejemplos de los apartados anteriores, para crear un archivo se ha instanciado un flujo de salida y se ha pasado una cadena con el nombre del archivo. Sin embargo, todos los constructores de clases de flujo de salida que esperan un archivo pueden recibir un objeto FILE con el nombre del archivo 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 argumento.

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.

Algunos ejemplos:

File mi Archivo = 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");
Crea el objeto otro *con el archivo Enseres.dat que está en la misma carpeta que*

COCINA;

File dir = new File ("C: \JAVA \EJERCICIOS") ;
File fil = new File(dir,"Complejos.dat");
Crea el objeto dir con un directorio o ruta absoluta. A continuación crea el objeto fil con el archivo Complejos. dat que está en la ruta especificada por el objeto dir.

Observación: La ventaja de crear objetos File con el archivo que se va a procesar es que podemos hacer controles previos sobre el archivo u obtener información sobr él..

Información sobre un archivo:

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

public boolean canWrite() *Devuelve true si se puede escribir en él, caso contrario 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 su tamaño en bytes.* Si *es directorio devuelve cero.*

public long lastModified() *Devuelve la hora de la última modificación.*

Un ejemplo usando class File: Chequearemos algunos de los métodos que acabamos de enumerar.

```
import java.io.*;
class Atributos {
  public void demo(){
    File miObj;         // Mi objeto
    BufferedReader entrada = new BufferedReader(
      new InputStreamReader(System.in));

    //Construyo entrada, objeto tipo BufferedReader,
    //su constructor necesita de otro objeto, del tipo
    //InputStrearnReader, cuyo parametro debe ser de
    //tipo System.in (texto)

    String cd;
    System.out.println("Informe nombre de archivo");
    System.out.println("o bien un directorio...");
    System.out.println("Termina con línea vacía. ");
    try{
      do{
         cd = entrada.readLine();
         miObj = new File(cd);
         if (miObj.exists()){
            System.out.println(cd + " existe !!!");
            if (miObj.isFile()){
               System.out.println(" Es un archivo,");
               System.out.println(" su tamaño: " + miObj.length());
            }
            else if (miObj.isDirectory())
                    System.out.println(" Es un directorio. ");
```

```
Informe nombre de archivo
o bien un directorio...
Termina con línea vacía.
d:\tymos
 No existe !
atributos.class
 atributos.class existe !!!
 Es un archivo,
 su tamaño: 1629
atributos.java
 atributos.java existe !!!
 Es un archivo,
 su tamaño: 1357
e:\tymos\catedras
 e:\tymos\catedras existe !!!
 Es un directorio.

Process Exit
```

```
                else
                  System.out.println(" Existe, desconocido...");
          }  //    verdadero de if (miObj.exists())
          else
             if (cd.length() > 0)
                 System.out.println(" No existe !");
       } while (cd.length() > 0);
     }
     catch(IOException e){System.out.println("Excepción " + e.getMessage());}
   } //demo

   public static void main(String[] args){
       Atributos atr = new Atributos();
       atr.demo();
   }
}
```

mas métodos de File

public String getName() // retorna *nombre del archivo o del directorio del objeto*
public String getpath() *// Devuelve la ruta relativa al directorio actual.*
public String getAbsolutePath() *// Devuelve la ruta completa del archivo o directorio.*
public boolean setReadOnly() *// Marca el archivo como de sólo lectura.*
public boolean delete() *// Elimina el archivo o directorio (debe estar vacío)*
public boolean renameTo(File nuevo) Renombra objeto File a nuevo.
public boolean mkdir() *// Crea el directorio con el que se ha creado el objeto.*
public String[] list() *// Devuelve un array de cadenas, contiendo elementos (archivo o directorio) pertenecientes al directorio en el que se ha inicializado el objeto.*

Archivos Secuenciales

La organización de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento, o también se define la organización como la forma en que se estructuran los datos en un archivo. En general se consideran tres organizaciones fundamentales:

· Organización secuencial.
· Organización directa o aleatoria.
· Organización secuencial indexada.

Un archivo con organización secuencial es una sucesión de registros almacenados consecutivamente sobre el soporte externo, de tal modo que para acceder a un registro *n* dado es obligatorio pasar por los n-l registros que le preceden.

En un archivo secuencial los registros se insertan en el archivo en orden de llegada, es decir, un
registro de datos se almacena inmediatamente a continuación del registro anterior.

Las operaciones básicas que se permiten en un archivo secuencial son: *escribir su contenido, aña*
dir un registro (al final del archivo) y *consultar registros.*

Java procesa los archivos como secuencias de bytes, no determina la estructura del archivo. El programador es el que determina el tipo de tratamiento del archivo, si va a ser secuencial o de acceso directo. Primero debe especificar el concepto de

registro según los datos que se van a almacenar; Java tampoco contempla la estructura de registro. Los campos en que se descompone cada registro, según el programador determine, se han de escribir uno a uno. Además, la operación de lectura del archivo creado tiene que hacerse de forma recíproca, si por ejemplo el primer campo escrito es de tipo entero, al leer el método de lectura debe ser para un entero. Cuando se terminan de escribir todos los reistros se cierra el flujo (close())

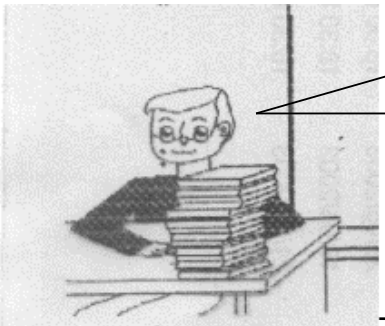
Creación del archivo secuencial Corredores.dat

El proceso de creación de un archivo secuencial es también secuencial, los registros se almacenan consecutivamente en el mismo orden en que se introducen.
El siguiente programa va a crear un archivo secuencial en el que se almacenan registros relativos a los corredores que participan en una carrera.

```
import java.io.*;
import In;
class Corredor{
      // Datos del corredor
      protected String nombre;
      protected int edad;
      protected char categoria;
      protected char sexo;
      protected int minutos;
      protected int segundos;

      // Datos del archivo
      protected File file;
      protected FileOutputStream miArch;
      protected DataOutputStream salida;
      protected BufferedReader entrada;

      public Corredor(){            // Constructor
            miArch  = null;   // Inicializando re
            entrada = null;
            salida    = null;
            file        = nul;
      }

      public boolean leerDatos()throws IOExcepti
            System.out.print("Nombre: ");
            nombre = In.readLine();
            if (nombre.length() > 0){
                  System.out.print("Edad: ");
                  edad = In.readInt();
                  if (edad >= 40) categoria = 'V
                  else              categoria =
                  System.out.print("Sexo,(M,F):
                  sexo = In.readChar();
                  System.out.print("Minutos:  ")
                  minutos = In.readInt();
                  System.out.print("Segundos: ")
                  segundos = In.readInt();
                  return true;
            }
            else return false;
      }
      // método para escribir en el flujo
```

```
Generando archivo de corredores

Nombre archivo: Corredores
(Ok: Archivo creado)

Informe datos del corredor
Para terminar, Nombre:<ENTER>

Nombre: Primero
Edad: 22
Sexo,(M,F): M
Minutos:  20
Segundos: 33
Nombre: Segundo
Edad: 21
Sexo,(M,F): M
Minutos:  22
Segundos: 50
Nombre: Tercero
Edad: 25
Sexo,(M,F): M
Minutos:  22
Segundos: 55
Nombre: Cuarto
Edad: 33
Sexo,(M,F): F
Minutos:  40
```

```
	public void grabarDatos () throws IOException{
		salida.writeUTF(nombre);
		salida.writeInt(edad);
		salida.writeChar(categoria);
		salida.writeChar(sexo);
		salida.writeInt(minutos);
		salida.writeInt(segundos);
	}

	public void crearArchivo(){
		System.out.println("Generando archivo de corredores");
		System.out.print("Nombre archivo: ");
		String nomeArch = In.readLine();
		nomeArch+=".dat";
		file    = new File(nomeArch);       //    Instanciando objeto File
		if (file.exists())
			System.out.println("(Ok: Archivo creado)");
		try{
			miArch = new FileOutputStream(file);
			salida = new DataOutputStream(miArch);
			entrada = new BufferedReader(
					new InputStreamReader(System.in));
		}
		catch (IOException e){System.out.println("Excepción creando archivo");}
		System.out.println("Informe datos del corredor");
		System.out.println("Para terminar, Nombre:<ENTER>");
		ile = new File("Corredores.dat");
		try{
			boolean mas = true;
			miArch = new FileOutputStream(file);
			salida = new DataOutputStream(miArch);
			BufferedReader entrada = new BufferedReader(
					new InputStreamReader(System.in));
			while (mas){
				mas = leerDatos();
				if (mas) grabarDatos();
			}
			miArch.close();
		}
		catch       (IOException       e){System.out.println("Excepción       de
Entrada/Salida");}
	}

	public static void main(String[] args){
		Corredor corr = new Corredor();
		corr.crearArchivo();
	}
}
```



Profe, Ud. disculpe, .pero esa clase Corredores está muy incompleta… Mínimamente debería tener comportamiento para leer el archivo y satisfacer alguna consulta. Sugiero, por ejemplo, poder informar cuales y cuantos corredores empatan en tantos minutos …

No me diga. Digamos que es una clase que es apenas un comienzo, está todo por hacer … Haremos lo que Ud pide, ya. Hum… Solo que en vez de modificar Corredores, la extendemos en una nueva clase que incorpore lo que Ud pide. Y, según opción del usuario, podremos crear el archivo o consultarlo … Le parece bien?

Consulta de un archivo secuencial

El proceso de consulta de una información en un archivo de organización secuencial se debe efectuar obligatoriamente en modo secuencia!. Por ejemplo, si se desea consultar la información contenida en el registro 50, se deberán leer previamente los 49 registros que le preceden. En el caso del archivo Corredores, si se desea buscar el tiempo de carrera de un participante del que se conoce su nombre y edad, será necesario recorrer todo el archivo desde el principio hasta encontrar el registro buscado o leer el último registro (fin de archivo).

En Java, para leer un archivo secuencial es necesario conocer el diseño que se ha hecho de cada registro, cómo se han escrito los campos en el proceso de creación. Los métodos que se utilicen para leer tienen que corresponderse con los métodos que se utilizaron para escribir. Así, si se escribió una cadena con el método writeUTF (), se ha de leer con readUTF (); si se escribió un entero con writeInt () , se leerá con readInt () , y así con cada tipo de dato.

```
import java.io.*;
import Corredor;
import In;
class CorreCaminos extends Corredor{
      protected char opcion;
      protected int minutes;
      // Datos del archivo
      FileInputStream  miArch;
      DataInputStream entrada;

      public CorreCaminos(){
            super();
      }

      public      boolean existeArchivo(){
            System.out.println("Consultando archivo de corredores");
            System.out.print("Nombre archivo: ");
            String nomeArch = In.readLine();
            nomeArch+=".dat";
            file    = new File(nomeArch);       //    Instanciando objeto File
            if (!file.exists()){
                  System.out.println("No encuentro "
                  return false;
            }
            try{
                  miArch = new FileInputStream(file)
                  entrada = new DataInputStream(miAr
            }
            catch (IOException e){
                  System.out.println("Excepción crea
                  return false;
            }
            return true;
      }

      public void consulta(){
```

```
Por favor, opte:

Crear  archivo: 1

Consultarlo . . 2

Salir . . <Enter>2
Consultando     archivo     de
corredores

Nombre archivo: Corredores
Consulta por empatados

Cuantos minutos ? 22
Corredores empatados
Segundo , 22 , 50
Tercero , 22 , 55
lectura terminada
```

```
            System.out.println("Consulta por empatados");
            while (true){
                  System.out.print("Cuantos minutos ? ");
                  minutes = In.readInt();
                  if (minutes > 0)break;
            }
            System.out.println(this);
      }

      public String toString(){
            boolean finArch = false;
            int cont = 0, cEmp = 0;
            String aux = "Corredores empatados\n";
            while (!finArch){
                  try{
                        nombre            = entrada.readUTF();
                        edad              = entrada.readInt();
                        categoria   = entrada.readChar();
                        sexo        = entrada.readChar();
                        minutos           = entrada.readInt();
                        segundos    = entrada.readInt();
                  }
                  catch(EOFException eof){finArch = true;}
                  catch(IOException io){aux+="Anomalia al procesar flujo";}
                  if (!finArch){
                        cont++;
                        if (minutos == minutes){
                              aux+=nombre+" , "+minutos+" , "+segundos+"\n";
                              cEmp++;
                        }
                  }
            }
            aux+= "lectura terminada \n";
            aux+="Leidos "+cont+", empat. "+cEmp+
            return aux;
      }

      public void queHago(){
            System.out.println("Por favor, opte:'
            System.out.println("Crear  archivo: 1");
            System.out.println("Consultarlo . . 2");
            System.out.print("Salir . . <Enter>");
            opcion = In.readChar();
            if (opcion == '1'){     // Debemos crear el archivo
                  crearArchivo();
            }
            if (opcion == '2'){
                  if (existeArchivo())
                        consulta();
            }
      }           //    public void queHago

      public static void main(String[] args){
            CorreCaminos corre = new CorreCaminos();
            corre.queHago();
      }
}
```

```
Por favor, opte:

Crear  archivo: 1

Consultarlo . . 2
Salir . . <Enter>2
Consultando     archivo     de
corredores
Nombre archivo: CorreCaminos
```

Actualización de un archivo secuencial

La actualización de un archivo con organización secuencial supone añadir nuevos registros, modificar datos de registros existentes o borrar registros; es lo que se conoce como *altas, modificaciones* y *bajas.*

El proceso de dar de alta un determinado registro o tantos como se requiera se puede hacer al inicializar el flujo de la clase FileOutputStream, pasando true como segundo argumento al constructor (el primer argumento es el archivo). De esta forma, el nuevo registro que se da de alta se añade al final del archivo.
Así, por ejemplo, para añadir nuevos registros al archivo Corredores*:*

```
File f = new File("Corredores.dat");
FileOutputStream mf = new FileOutputStream(f,true);
DataOutputStream fatI = new DataOutputStream(mf) ;
```

Para dar de baja a un registro del archivo secuencial se utiliza un archivo auxiliar, también secuencial. Se lee el archivo original registro a registro y en función de que coincida o no con el que se quiere dar de baja se decide si el registro debe ser escrito en el archivo auxiliar. Si el registro se va a dar de baja, se omite la escritura en el archivo auxiliar. Si el registro no se da de baja, se escribe en el archivo auxiliar. Una vez que termina el proceso completo con todo el archivo, se tienen dos archivos: el original y el auxiliar. El proceso de baja termina cambiando el nombre del archivo auxiliar por el del original. Los flujos que se deben crear son:

```
File fr = new File("Original");
FileInputStream fo = new FileInputStream(fr) ;
DataInputStream forg = new DataInputStream(fo) ;

// para el archivo auxiliar
File fx = new File("Auxiliar");
FileOutputStream fa = new FileOutputStream(fx) ;
DataOutputStream faux = new DataOutputStream(fa) ;
```

Con la llamada a los métodos de la clase File: delete () y renameTo () , para eliminar el archivo original una vez procesado y cambiar de nombre el archivo auxiliar, se termina el proceso de dar de baja.

Las actualizaciones de registros de un archivo secuencial se pueden hacer siguiendo los mismos pasos que dar de baja. Los registros del archivo original se leen y se escriben en el archivo auxiliar, hasta alcanzar el registro a modificar. Una vez que se alcanza éste, se cambian los datos o campos deseados y se escribe en el archivo auxiliar. El archivo original se procesa hasta leer el último registro; se termina borrando el archivo original y cambiando el nombre del auxiliar por el nombre del original.
NOTA: Las actualizaciones de un archivo secuencial con movimientos de altas, bajas y modificaciones (ABM) requieren que ambos archivos estén ordenados idénticamente, por los mismos campos. La lógica del programa que trata estas actualizaciones no es trivial. Normalmente esta problemática se resuelve con una base de datos y estas actualizaciones se declaran en en lenguaje SQL (Update myBase …)

Archivos de Acceso directo (Random Access File)

Un archivo es de acceso aleatorio o directo cuando cualquier registro es directamente accesible mediante la especificación de 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 la posición del registro se puede situar el *puntero* de lectura/escritura en la posición donde comienza el registro y a partir de esa posición realizar la operación de lectura o escritura. Muy distinto el caso del acceso a un registro n en la organización secuencial, supone leer (recorrer) los n -1 registros anteriores.

Las operaciones que se realizan con los archivos de acceso directo son las usuales: creación, consulta, altas, 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; en particular para los campos de tipo String se debe prever el máximo número de caracteres, además si se escribe en formato UTF se añaden dos bytes más (en los dos bytes se guarda la longitud de la cadena). Los campos de tipo primitivo tienen longitud fija: char, dos bytes; int, cuatro bytes; double, ocho bytes...

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, RandomAccessFile 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.

Creación de un objeto RandomAccessFile

La clase tiene dos constructores con dos argumentos, el primero es el archivo y el segundo una cadena con el *modo* de abrir el flujo.

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 RandomAccessFile(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..

En cuanto al modo de apertura puede ser de dos formas:
"**r**" Modo de sólo lectura. Únicamente se pueden realizar de lectura de registros.
"**rw**" Modo lectura/escritura. Permite operaciones de entrada/salida.

Por ejemplo, se tiene el archivo Libros. dat y se quiere únicamente consultar, con acceso directo, los flujos a crear:

```
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.

En este otro ejemplo se abre un flujo para cualquier operación de entrada o de salida:

```
try{ RandomAccessFile dlib = new RandomAccessFile("archSal.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 cual 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; si por ejemplo el puntero se encuentra en la posición n y se lee un campo entero (cuatro bytes) y un campo double (ocho bytes), la posición del *puntero* del archivo será n+12 .

public long getFilePointer() throws IOException: Este método de la clase RandomAccessFile devuelve la posición actual del *puntero del archivo.* Devuelve un entero de tipo long que representa el número de bytes desde el inicio del archivo hasta la posición actual. Su declaración:

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("archSal.dat","rw");
System.out.println("Posición del puntero: " + acc.getFilePointer());
```

public void seek(long n) throws IOException: Este método desplaza el puntero del archivo n bytes tomando como origen el del archivo (byte 0). Es el método más utilizado debido a que sitúa el puntero en una posición determinada.

public long length() throws IOException: Con este método se obtiene el tamaño actual del archivo, el número de bytes que ocupa o longitud que tiene.

Por ejemplo, si queremos posicionar el puntero al final de un archivo:

```
RandomAccessFile acc = new RandomAccessFile("archSal.dat","rw");
acc.seek(acc.length()) ;
```

Creando un archivo directo

El ejercicio que a continuación se escribe muestra los pasos a seguir para crear un archivo, que no existe, de acceso directo o aleatorio.
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 están identificados por un número de orden.Se quiere guardar la muestra en un archivo con acceso directo, siendo el número de orden de cada casa el número de registro.*
Lo primero a determinar va a ser la longitud que va a tener cada registro lógico, que depende de los campos y del ancho de cada campo. Así, consideramos como primer campo numOrden, de tipo in t y por tanto cuatro bytes; el campo nombreCasa, como máximo 40 caracteres, se va a escribir en formato UTF, que añade dos bytes más, por lo que ocupa un máximo de 82 bytes. Las coordenadas son dos números reales de tipo double, y por tanto son 8+8 bytes. En total cada registro tiene una longitud de

4+82+8+8=102.

El archivo va a tener el nombre de ***miCountry.dat***. Se ha de crear un objeto File con el nombre del archivo y la ruta. El flujo (objeto de 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, por lo que en un principio el máximo de registros es 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, que podrán ser ocupados posteriormente.

```
import java.io.*;
class MiCasa{
      protected int numOrden;
      protected String nombreCasa;
      protected double x, y;
      protected final int TAM = 102;

      public MiCasa(){
            nombreCasa = new String();
            numOrden = 0;
            x = y = 0.0;
      }

      public boolean altaReg(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 grabar alta");
                  flag = false;

            }
            return flag;
      }

      public int getTamanho() {return TAM;}

      // desplazamiento en bytes del registro n
      public long desplazamiento(int n) {return TAM*(n-1);}
}

// clase principal
import MiCasa;
import In;
import java.io.*;
class CreaCountry{
      File af;
      RandomAccessFile arch;
      MiCasa reg;
      final int MAXREG = 99;

      public CreaCountry(){
            try{
```

```
Para terminar, Número de
orden = 0

Número de orden:12
Nombre: Luna Azul
Coordenadas (x,y):12 27
Número de orden:11
Nombre: Sol violeta
Coordenadas (x,y):13 27
Número de orden:15
Nombre: Marte Rojo
Coordenadas (x,y):14 27
Número de orden:16
Nombre: Rojo Atardecer
Coordenadas (x,y):15 27
Número de orden:
```

```
            af = new File("miCountry.dat");
            arch = new RandomAccessFile(af,"rw");
            reg = new MiCasa();

      }
      catch(IOException e){
            System.out.println("No se puede abrir el flujo."+e.getMessage());
            System.exit(1);
      }
}

void archVacio(){
      try{
            reg = new MiCasa(); // crea registro vacío
            for (int k = 1; k <= MAXREG; ++k){
                  arch.seek(reg.desplazamiento(k));
                  reg.altaReg(arch);
            }
      }
      catch(IOException e){
            System.out.println("Excepción al mover puntero.del archivo");
            System.exit(1);
      }
}

void procesoEntrada(){
      BufferedReader entrada = new BufferedReader(
            new InputStreamReader(System.in));
      System.out.println("Para terminar, Número de orden = 0");
      try{
            int nr;
            double x, y;
            String cd;
            boolean masRegistros = true;
            // bucle para la entrada
            while (masRegistros){
                  do{               // Ciclo de consistencia
                        System.out.print("Número de orden:");
                        nr = In.readInt();
                  }while (nr<0 || nr>MAXREG);
                  if (nr > 0 ){     // no es clave de salida
                        reg.numOrden = nr;
                        System.out.print("Nombre: ");
                        cd = In.readLine();
                        // se asegura un máximo de 40 caracteres
                        reg.nombreCasa
                              = cd.substring(0,Math.min(40,cd.length()));
                        do{
                              System.out.print("Coordenadas (x,y):");
                              x = In.readDouble();
                              y = In.readDouble();
                        }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.altaReg(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");}
      }

      public static void main(String []args){
            CreaCountry country = new CreaCountry();
            country.archVacio();
            country.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, por tanto en cierto modo es un proceso secuencia!. 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, esa 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 MiCasa 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. Lo que hacemos, en realidad es **extender MiCasa en MiCasona**.

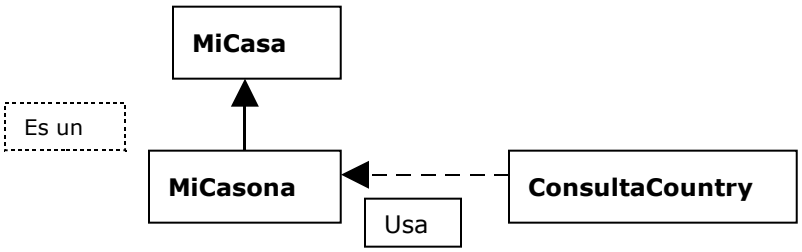
Luego creamos una nueva clase, ConsultaCasa, en la que se definen los métodos de consulta ya citados.

```
import MiCasa;
import java.io.*;
class MiCasona extends MiCasa{

      public MiCasona(){super();}

      public boolean leerReg(RandomAccessFile flo) throws EOFException {
            boolean 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
import java.io.*;
import In;
class ConsultaCountry{
   private RandomAccessFile archivo;
   private void mostrar(MiCasona reg){
      System.out.println(reg.nombreCasa +
         " ubicada en las coordenadas (" + (float)reg.x + ", " + (float)reg.y +")");
   }

   public ConsultaCountry(File f) throws IOException{//    constructor
      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{    // todo el archivo
            if (archivo != null){
                  try{
                        int n = 0;
                        MiCasona reg = new MiCasona();
                        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{// Un reg. determinado
            if (archivo != null){
                  MiCasona reg = new MiCasona();
                  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.");
            }
      }
      public static void main(String []args){
            File f = new File("MiCountry.dat");
            ConsultaCountry clta = null;
            int opc = 1;
            BufferedReader entrada = new BufferedReader(
                  new InputStreamReader(System.in));
            try{
                  System.out.println("\n Consultas al archivo" +
                  f.getName());
                  clta = new ConsultaCountry(f);
                  while (opc != 5){
                        System.out.println("1.Muestra registros activos.") ;
                        System.out.println("2.Muestra un registro determinado.");
                        System.out.print  ("5.Se sale de la aplicación.");
                        do{
                              opc = In.readInt();
                        }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 = In.readInt();
                                    clta.consultaReg(nreg);
                        }
                  }
            }
            catch(IOException e){
                  System.out.println("\n Excepción duranteconsulta:" +
                        e.getMessage() + " Termina la aplicación.");
            }
      }
}
```

```
Consultas al archivo MiCountry.dat
Flujo abierto en modo lectura, tamaño: 10018

1.Muestra registros activos.
2.Muestra un registro determinado.
5.Se sale de la aplicación.1

Sol violeta ubicada en las coordenadas (13.0,
27.0)
Luna Azul ubicada en las coordenadas (12.0, 27.0)
Marte Rojo ubicada en las coordenadas (14.0, 27.0)
Rojo Atardecer ubicada en las coordenadas (15.0,
27.0)
Se alcanza el fin del archivo en la consulta

1.Muestra registros activos.
2.Muestra un registro determinado.
5.Se sale de la aplicación.2

Número de registro: 1
Registro 1 no dado de alta.

1.Muestra registros activos.
2.Muestra un registro determinado.
```

El diagrama de clases de la consulta es:



Flujos de tipo objeto

El uso de registros de longitud fija es una buena elección cuando necesite almacenar datos del mismo tipo. Sin embargo, los objetos creados en un programa OOP raramente lo serán. Por ejemplo, puede tener un array llamado staff que, nominalmente, es un array de registros Empleados pero que contiene objetos que son instancias de una clase hija como Gerentes.

Si tenemos que guardar archivos que contengan este tipo de información, primero tendremos que almacenar el tipo de cada objeto y después los datos que definen el estado actual del mismo. Cuando recuperemos después esta información desde el archivo, tendremos que:

· Leer el tipo del objeto.
· Crear un objeto en blanco de ese tipo.
· Rellenarlo con los datos almacenados en el archivo.

Esto es posible hacerlo "artesanalmente", y así es como se hacía. Pero ya no es necesario. Sun Microsystems desarrolló un potente mecanismo que permite efectuar esta operación con mucho menos esfuerzo. Como verá muy pronto, este mecanismo, llamado **serialización de objetos**, casi automatiza por completo lo que antes resultaba ser un proceso muy tedioso.

Almacenar objetos de distinto tipo

Para guardar un objeto, primero tiene que abrir un objeto ObjectOutputStream:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("Empleados.dat"));
```

Ahora, para efectuar el almacenamiento, basta con usar el método writeObject de la clase ObjectOutputStream:

```
Empleados tomas = new Empleados(… Datos de Tomas …);
Gerentes bill = new Gerentes(… Datos de Bill …);
out.writeObject(tomas);
out.writeObject(bill);
```

Para leer los objetos, primero se debe obtener un objeto ObjectlnputStream:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("Empleados.dat"));
```

Después, los objetos se recuperan en el mismo orden en el que fueron escritos mediante el método readObject:

```
Empleados e1 = (Empleados)in.readObject();
Empleados e2 = (Empleados)in.readObject();
```

Cuando se recuperan objetos, es muy importante saber cuántos de ellos se han guardado, su orden y sus tipos. Cada llamada a readObject lee otro objeto de tipo Object. Por tanto, tendrá que moldearlo al tipo que usted necesite.

Si no necesita el tipo exacto, o no se acuerda de él, puede moldearlo a cualquier superclase o, incluso, dejarlo como Object. Por ejemplo, e2 es una variable Empleados incluso aunque haga referencia a un objeto Gerentes. Si necesita obtener de forma dinámica el tipo de un objeto, puede usar el método getClass usado un poco mas adelante en este apunte.

Con los métodos writeObject/readObject, sólo es posible leer objetos, no números. Para efectuar estas operaciones, debe usar otros métodos como writeInt/readInt o writeDouble/readDouble (las clases de flujo de tipo objeto implementan las interfaces DataInput/DataOutput). Desde luego, los números que se encuentren dentro de objetos (como el campo salario de un objeto Empleados) se guardan y recuperan de manera automática. Recuerde que, en Java, las cadenas y los arrays son objetos y que, por consiguiente, pueden ser manipulados con los métodos writeObject/ readObject.

Sin embargo, existe un agregado que es necesario hacer en cualquier clase que deba trabajar con flujos de tipo objeto. Dicha clase debe implementar la interfaz Serializable:

```
class Empleados implements Serializable { . . . }
```

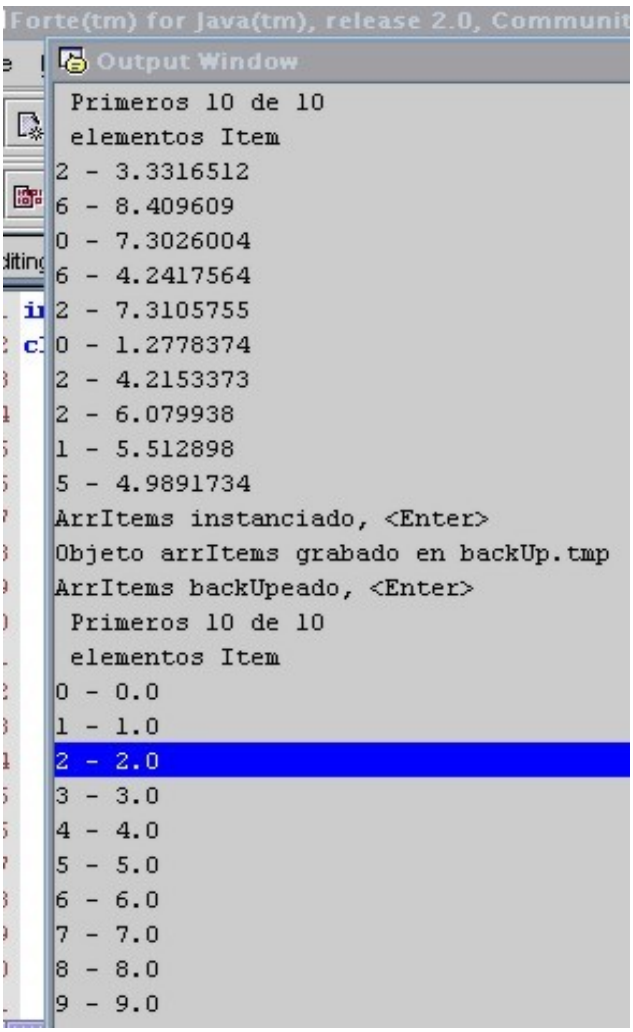
La interfaz Serializable **no tiene métodos,** por lo que no es necesario cambiar la clase de ninguna forma.

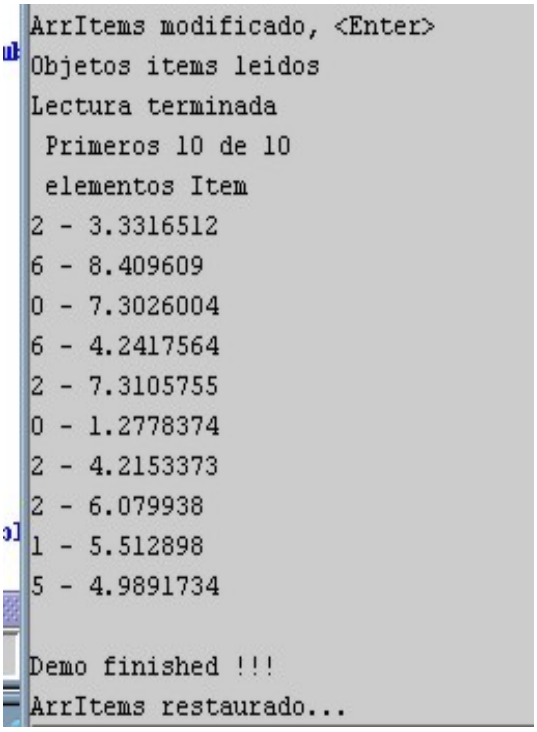
Hagamos un primer ejemplo; en él unicamente pretendemos grabar un objeto, luego modificarlo en memoria y para terminar **recuperarlo** (Leerlo desde disco) demostrando que hemos vuelto a sus valores iniciales. Para ello utilizaremos la clase **ObjItems,** una clase de prueba cuyo objetivo es la grabación/lectura de objetos, su único atributo será un objeto de la clase ArrItems, y sus métodos permiten un tratamiento demostrativo del objetivo propuesto.

Para entender en forma simple lo que hacemos, siga el main(). Allí verá que primero instanciamos un array de 10 items desordenado, lo grabamos, luego lo modificamos ordenandolo por código y finalmente deshacemos esta modificación volviendo los datos a lo que tenemos en el archivo de objetos. El capture de la ejecución muestra estos pasos y el detalle está en los metodos de **class ObjItems.**

```
import java.io.*;
class ObjItems implements Serializable{
   private ArrItems arrItems;
   public ObjItems(int tam, char tipo){
      arrItems = new ArrItems(tam,tipo);
   }
   public void demoGrab() throws IOException{
      FileOutputStream arch=new FileOutputStream("backUp.tmp");
      ObjectOutputStream objOut=new ObjectOutputStream(arch);
      if(objOut!=null){
         objOut.writeObject(arrItems);
         objOut.close();
         System.out.println("Objeto arrItems grabado en backUp.tmp");
      }else System.out.println("Objeto objOut no instanciado (???)");
  }
  public void demoMod(){
     for(int i=0;i<arrItems.talle;i++){
        arrItems.item[i].setCodigo(i);  arrItems.item[i].setValor((float)i);}
     System.out.println(arrItems);
  }
  public void demoLect() throws IOException{
     FileInputStream arch=new FileInputStream("backUp.tmp");
     ObjectInputStream objIn=new ObjectInputStream(arch);
     if(objIn!=null)
        try{System.out.println("Objetos items leidos");
           while(true)  arrItems = (ArrItems)objIn.readObject();
        }catch (EOFException eof){
              objIn.close();
              System.out.println("Lectura terminada");
              System.out.println(arrItems);
        }catch (ClassNotFoundException nfe){
              System.out.println("ClassNotFoundException");
        }finally {System.out.println("Demo finished !!!");}
  }
  public static void main(String[] args) throws IOException {
            char x;  ObjItems grbIt = new ObjItems(10,'R');
            System.out.println("ArrItems instanciado, <Enter>");
            x = In.readChar();
            grbIt.demoGrab();
            System.out.println("ArrItems backUpeado, <Enter>");
            x = In.readChar();
            grbIt.demoMod();
            System.out.println("ArrItems modificado, <Enter>");
            x = In.readChar();
```

```
            grbIt.demoLect();
            System.out.println("ArrItems restaurado...");
        }
}       //      class ObjItems
```



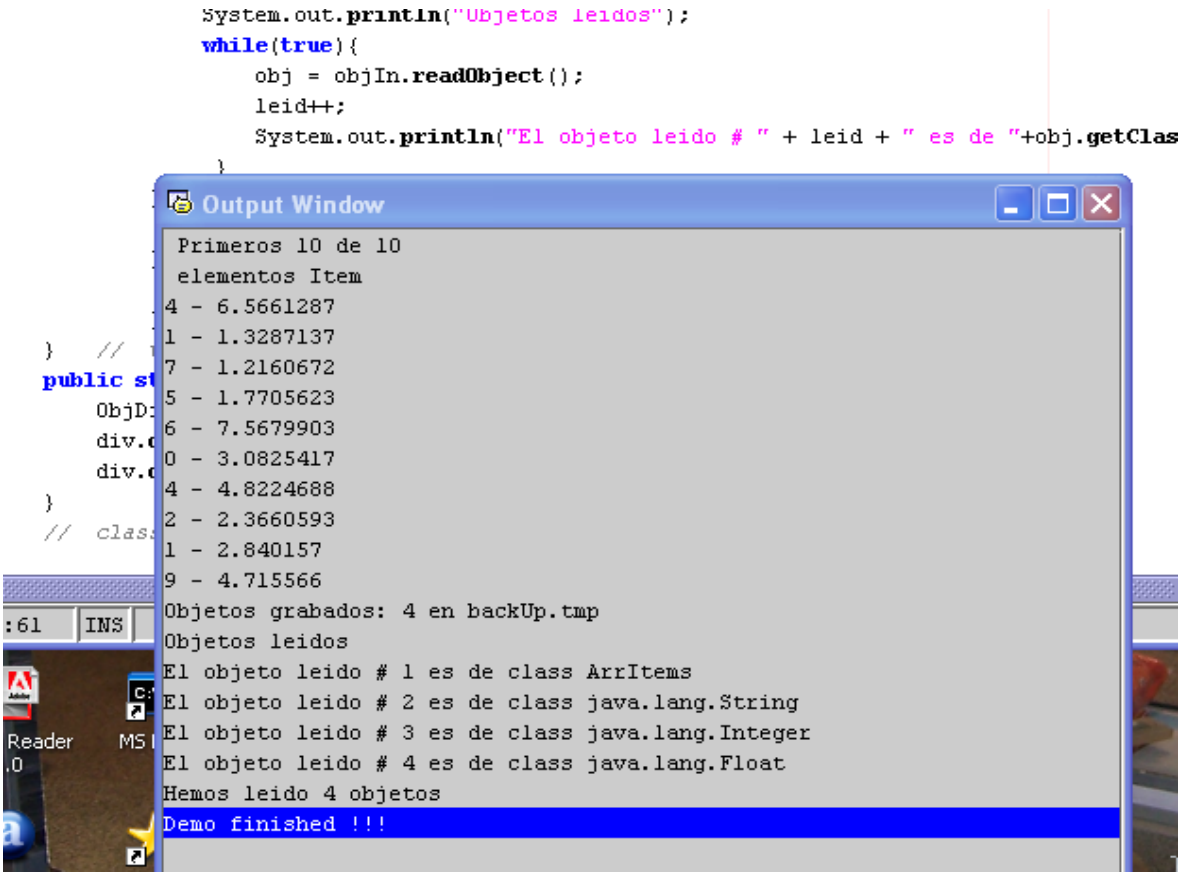


Vemos que el objeto arrItems contiene sus valores originales. Eso significa que lo hemos grabado y leido con éxito. (En realidad grabamos y leemos un único objeto ArrItems, constituido por objetos Item, del mismo tipo).

Vamos a un segundo ejemplo. Tratando objetos diversos: grabando, leyendo, reconociendo...
En este ejemplo Ud verá que grabamos objetos de distintos tipos, absolutamente diversos. Luego los leemos en una rutina totalmente genérica. Como no deseamos tratarlo de una manera específica los dejamos nomás en la clase Object. Y como sólo nos interesa saber su clase , pues usamos el método getClass() de la clase Object.

```
import java.io.Serializable;
import java.io.*;
class ObjDiv implements Serializable{
   private ArrItems arrItems;
   private String cadena = "Lo que es verdad a la luz de la lampara...";
   private Integer entero;
   private Float   decimal;
   private int grab =0, leid =0;
```

```
   public ObjDiv(int tam, char tipo){
      arrItems = new ArrItems(tam,tipo);
      entero   = new Integer(10);
      decimal  = new Float(10.55);
   }
   public void demoGrab() throws IOException{
      FileOutputStream arch=new FileOutputStream("backUp.tmp");
      ObjectOutputStream objOut=new ObjectOutputStream(arch);
      if( objOut!=null){
         objOut.writeObject(arrItems);   grab++; objOut.writeObject(cadena);     grab++;
         objOut.writeObject(entero);      grab++; objOut.writeObject(decimal);    grab++;
         objOut.close();
         System.out.println("Objetos grabados: " + grab + " en backUp.tmp");
      }else System.out.println("Objeto objOut no instanciado (???)");
   }
   public void demoLect() throws IOException{
      FileInputStream arch=new FileInputStream("backUp.tmp");
      ObjectInputStream objIn=new ObjectInputStream(arch);
      Object obj;
      if( objIn!=null)
         try{
            System.out.println("Objetos items leidos");
            while(true){
               obj = objIn.readObject();
               leid++;
               System.out.println("El objeto leido # " + leid + " es de "+obj.getClass());
             }
         }catch (EOFException eot){ objIn.close();
            System.out.println("Hemos leido " + leid +" objetos");
         }catch (ClassNotFoundException nfe){
            System.out.println("Class Not FoundException ");
         } finally {System.out.println("Demo finished !!! ");}
   }  //  void demoLect()
   public static void main(String[] args) throws IOException {
      ObjDiv div = new ObjDiv(10,'R');
      div.demoGrab();
      div.demoLect();
   }
}  //  class ObjDiv
```

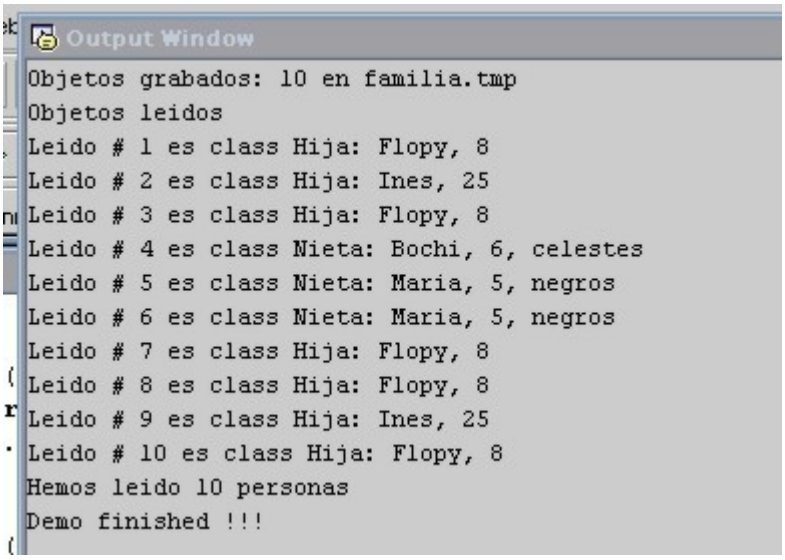


Y un último ejemplo, donde ademas incorporamos un tratamiento polimórfico a los objetos.

`Tratando objetos diversos: grabando, leyendo, reconociendo...`

```
import java.io.Serializable;
import java.io.*;
class Madre implements Serializable{
   protected String nombre;
   public Madre(String nome){nombre = nome;}
   public String toString(){return nombre;}
}
class Hija extends Madre implements Serializable{
   protected int edad;
   public Hija(String nome, int age){
      super(nome);
      edad = age;
   }
   public String toString(){
      return nombre+", "+edad;
   }
}
class Nieta extends Hija implements Serializable{
   protected String ojos;
   public Nieta(String nome, int age, String eyes){
      super(nome, age);
      ojos = eyes;
   }
   public String toString(){
      return nombre+", "+edad+", "+ojos;
   }
```

```
}
class ObjDivPol implements Serializable{
    private Madre persona[];
    private int grab =0, leid =0;
    public ObjDivPol(){
        persona = new Madre[5];
        persona[0] = new Madre("Dercilia");
        persona[1] = new Hija("Ines", 25);
        persona[2] = new Nieta("Bochi",6,"celestes");
        persona[3] = new Nieta("Maria",5,"negros");
        persona[4] = new Hija("Flopy",8);
   }
   public void demoGrab() throws IOException{
        FileOutputStream arch=new FileOutputStream("familia.tmp");
        ObjectOutputStream objOut=new ObjectOutputStream(arch);
        int i, ind;
        if( objOut!=null){
            for(i= 0;i<10;i++){
                ind = (int)(5*Math.random());
                objOut.writeObject(persona[ind]);
                grab++;
            }
            objOut.close();
            System.out.println("Objetos grabados: " + grab + " en familia.tmp");
        }else System.out.println("Objeto objOut no instanciado (???)");
    }
        public void demoLect() throws IOException{
        FileInputStream arch=new FileInputStream("familia.tmp");
        ObjectInputStream objIn=new ObjectInputStream(arch);
        Object obj;
        if( objIn!=null)
            try{
                System.out.println("Objetos leidos");
                while(true){
                    obj = objIn.readObject();
                    leid++;
                    System.out.println("Leido # " + leid + " es "+obj.getClass());
                    System.out.println(obj);
                 }
            }catch (EOFException eot){ objIn.close();
                System.out.println("Hemos leido " + leid +" personas");
            }catch (ClassNotFoundException nfe){
                System.out.println("Class Not FoundException ");
            } finally {System.out.println("Demo finished !!! ");}
    }  //  void demoLect()
    public static void main(String[] args) throws IOException {
        ObjDivPol divPol = new ObjDivPol();
        divPol.demoGrab();
        divPol.demoLect();
    }
}  //  class ObjDivPol
```



A continuación, completando esta unidad incluimos el trabajo del Ing. Valerio Frittelli, presentado en el cierre de los talleres de Algoritmos y Estructuras de Datos (Objetos, Java).

Recordando interfaces

Para que interactúen dos objetos, deben "conocer" los diversos mensajes que acepta cada uno, esto es, los métodos que soporta cada objeto. Si queremos **forzar** ese "conocimiento", el paradigma de diseño orientado a objetos pide que las clases especifiquen la *interfaz de programación de aplicación* (API, de *application programming interface)* o simplemente la *interfaz* que sus objetos presentan a otros objetos. Una interfaz se especifica como una definición de tipo y un conjunto de métodos para este tipo. En Java, esta especificación es aplicada por la máquina virtual, que requiere que los tipos de parámetros que se pasan a los métodos, sean del tipo especificado en la interfaz. A este requisito se le llama *tipeado fuerte.* El tener que definir interfaces para después hacer que esas definiciones sean obligatorias por tipificación fuerte es una carga para el programador, pero está compensada por las ventajas que proporciona porque impone el principio de encapsulamiento, y con frecuencia atrapa errores de programación que de otra forma pasarían inadvertidos.

Implementando interfaces

El elemento estructural principal de Java que impone una API es la *interfaz.* Una interfaz es un conjunto de declaraciones de método, **sin datos ni cuerpos**. Esto es, los métodos de una interfaz siempre están vacíos. Cuando una clase implementa una interfaz, debe implementar todos los métodos declarados en la interfaz. De esta forma, las interfaces imponen una especie de herencia, llamada *especificación,* donde se requiere que cada método heredado se especifique por completo.

Por ejemplo, supóngase que se desea crear un inventario de las antigüedades que poseemos, en categorías de varios tipos y de diversas propiedades. Se podría, por ejemplo, tratar de identificar algunos de los objetos como vendibles; la interfaz Vendible define cual es el comportamiento que obligatoriamente estos objetos deben satisfacer.

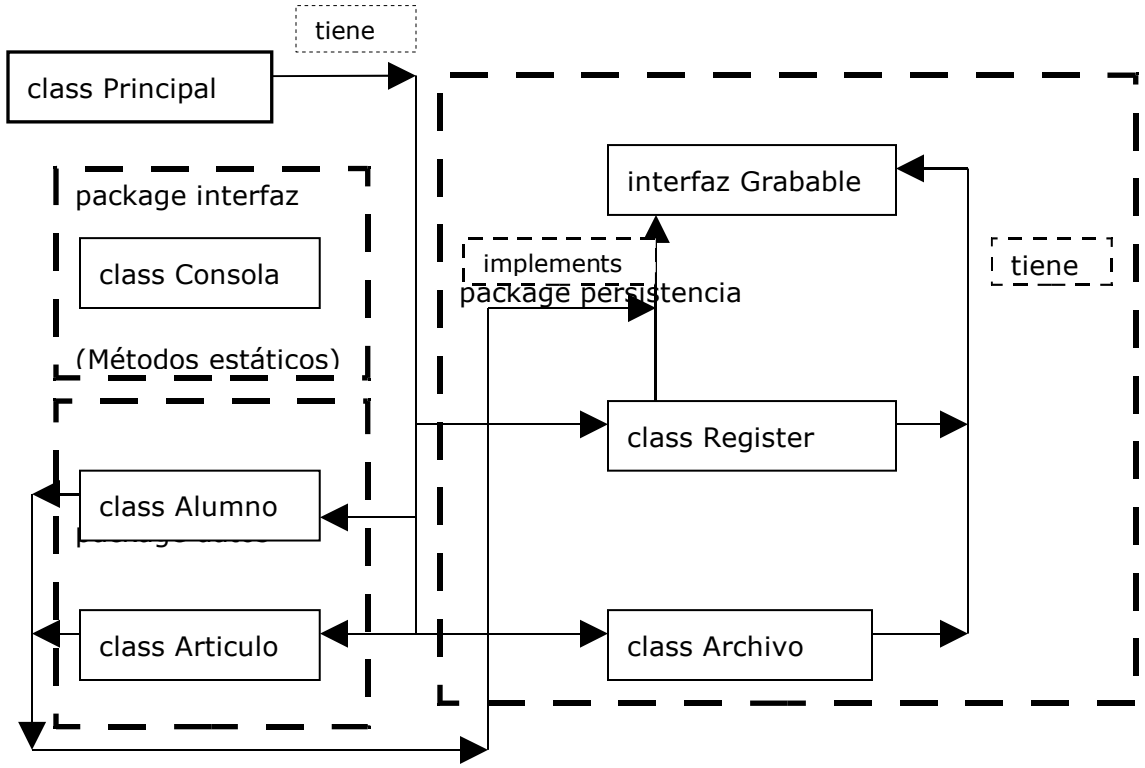
```
public interface Vendible {  // Interfaz para objetos que se pueden vender.
        public String description(); //  descripción del objeto
        public int listPrice();          // precio de lista
        public int lowestPrice();      // precio mínimo
}
```

A continuación se puede definir una clase concreta, Fotografias, que implementa la interfaz Vendible, indicando que se desea vender cualquiera de sus objetos; esta clase define el cuerpo a cada método de la interfaz Vendible. Además, agrega un método, isColor(), que específico para los objetos Fotografias.

```
public class Fotografias implements Vendible { // Clase para fotografías que se pueden vender
      private String descript;              // descripción de esta foto
      private int price;                    // el precio establecido
```

```
        private boolean color;                  // cierto si la foto es en colores
        public Fotografias(String desc, int p, boolean c) { //constructor
              descript = desc; price = p; color = c;
        }

         public String description() { return descript;}
         public int listPrice(){ return price;}
         public int lowestPrice() { return price/2;}
         public boolean isColor() { return color;}
}
```

El trabajo del Ing. V. Frittelli documenta extensamente "in situ" toda la codificación, por lo que no precisa de presentación adicional. Graficamos las relaciones de clases, interfaces y paquetes.




```
/**
 * Clase para contener al método main.
 *
 * @author Ing. Valerio Frittelli
 * @version Julio de 2005
 */
// import persistencia.*;
// import datos.*;
// import interfaz.*;
public class Principal
{
        private static Archivo m1, m2;
        private static Register reg;
        private static Alumno alu;
        private static Articulo art;

        /**
```

```
         * Muestra el contenido de un archivo (incluidos los registros marcados como borrados) en consola estandar
         */
        public static void mostrarTodo (Archivo m)
        {

            m.openForRead();
            while (!m.eof())
            {
                reg = m.leer();
                mostrarRegistro(reg);
            }
            m.close();
        }

        /**
         * Muestra el contenido del archivo (sólo los registros que no estén marcados como borrados) en consola estandar
         */
        public static void mostrar (Archivo m)
        {
            m.openForRead();
            while (!m.eof())
            {
                reg = m.leer();
                if(reg.isActive()) mostrarRegistro(reg);
            }
            m.close();
        }


       /**
        * Carga un legajo por teclado
        */
       public static void cargarLegajo( )
       {
            System.out.print("Ingrese el Legajo: ");
            int legajo = Consola.readInt();
            alu.setLegajo(legajo);
       }

       /**
        * Carga un registro de Alumno por teclado
        */
       public static void leerAlumno ( )
       {
            cargarLegajo();
            System.out.print("Ingrese el Nombre: ");
            String nombre = Consola.readLine();
            alu.setNombre(nombre);
            System.out.print("Ingrese el Promedio: ");
            float promedio = (float)Consola.readDouble();
            alu.setPromedio(promedio);
       }

       /**
        * Carga un codigo de Articulo por teclado
        */
       public static void cargarCodigo( )
       {
```

```
        System.out.print("Ingrese el Código: ");
        int codigo = Consola.readInt();
        art.setCodigo(codigo);
    }

    /**
     * Carga un registro de Articulo por teclado
     */
    public static void leerArticulo ( )
    {
        cargarCodigo();
        System.out.print("Ingrese la descripción: ");
        String nombre = Consola.readLine();
        art.setDescripcion(nombre);
    }


    /**
     * Muestra un registro por consola estandar
     */
    public static void mostrarRegistro (Register reg)
    {
      System.out.print(reg.getData().toString());  // que sería lo mismo que System.out.print(reg.getData())
      System.out.println("\tActivo?: " + reg.isActive());
    }


public static void main (String[] args)
{
    int x, op;

    try
    {
       m1 = new Archivo("Alumnos.dat", new Alumno());
       m2 = new Archivo("Articulos.dat", new Articulo());
    }
    catch(ClassNotFoundException e)
    {
       System.out.println("Error al crear los descriptores de archivos: " + e.getMessage());
       System.exit(1);
    }

    alu = new Alumno();
    art = new Articulo();

    do
    {
       System.out.println ("Opciones ABM de archivos");
       System.out.println ("1.  Alta de un registro de Alumno");
       System.out.println ("2.  Alta de un registro de Articulo");
       System.out.println ("3.  Baja de un registro de Alumno (lógica)");
       System.out.println ("4.  Baja de un registro de Articulo (lógica)");
       System.out.println ("5.  Modificacion de un registro de Alumno");
       System.out.println ("6.  Modificacion de un registro de Articulo");
       System.out.println ("7.  Listado de Alumnos");
       System.out.println ("8.  Listado de Articulos");
       System.out.println ("9.  Depuracion de Alumnos (bajas fisicas)");
       System.out.println ("10. Depuración de Artículos (bajas físicas)");
       System.out.println ("11. Salir");
```

```
System.out.print ("Ingrese opcion: ");
op = Consola.readInt ();
switch (op)
{
   case 1:
         System.out.println("Ingrese los campos del registro de Alumno: ");
         leerAlumno();
         m1.alta(new Register(alu));
         break;

   case 2:
         System.out.println("Ingrese los campos del registro de Artículo: ");
         leerArticulo();
         m2.alta(new Register(art));
         break;

   case 3:
         System.out.print("Ingrese el legajo del alumno a borrar: ");
         x = Consola.readInt();
         alu.setLegajo(x);
         m1.baja(new Register(alu));
         break;

   case 4:
         System.out.print("Ingrese el código del artículo a borrar: ");
         x = Consola.readInt();
         art.setCodigo(x);
         m2.baja(new Register(art));
         break;

   case 5:
         System.out.print("Ingrese los datos para modificar un registro de Alumno: ");
         leerAlumno();
         m1.modificar(new Register(alu));
         break;

   case 6:
         System.out.print("Ingrese los datos para modificar un registro de Artículo: ");
         leerArticulo();
         m2.modificar(new Register(art));
         break;

   case 7:
         System.out.println("Se muestra el archivo de Alumnos (incluyendo 'borrados')...");
         mostrarTodo(m1);
         break;

   case 8:
         System.out.println("Se muestra el archivo de Articulos (incluyendo 'borrados')...");
         mostrarTodo(m2);
         break;

   case 9:
         System.out.println("Se eliminan registros de Alumnos marcados...");
         m1.depurar();
         System.out.print("\nOperacion terminada...");
         break;

   case 10:
         System.out.println("Se eliminan registros de Articulos marcados...");
```

```
                    m2.depurar();
                    System.out.print("\nOperacion terminada...");
                    break;

            case 11:  ;
          }
        }
        while (op != 11);
    }
}
```

```
package interfaz;

/**
 * Clase para facilitar operaciones de carga por teclado en consola estándar
 * @author Instructores de CISCO System para su curso de Fundamentos de Java 1.1 - Modificado por Valerio Frittelli
 * @version Mayo de 2004
*/

public class Consola
{
   /*
    * 1) Todos los métodos de esta clase son estáticos, y por lo tanto pueden ser invocados sin tener que crear objetos de la clase. Es
    * suficiente con nombrar la clase al invocar el método:   int x = Consola.readInt();
    */

   /**
    * Lee un string desde teclado. El string termina con un salto de linea
    * @return el string leido (sin el salto de linea)
   */
   public static String readLine()
   {
     int ch;
     String r = "";
     boolean done = false;
     while (!done)
     {
           try
           {
                 ch = System.in.read();
                 if (ch < 0 || (char)ch == '\n') { done = true; }
                 else
                 {
                     if ((char)ch != '\r') { r = r + (char) ch; }
                 }
           }
           catch(java.io.IOException e)
           {
                 done = true;
           }
     }
     return r;
   }

   /**
    * Lee un integer desde teclado. La entrada termina con un salto de linea
    * @return el valor cargado, como un int
    */
```

```
   public static int readInt()
   {
     while(true)
     {
          try
          {
             return Integer.parseInt(readLine().trim());
          }
          catch(NumberFormatException e)
          {
             System.out.println("No es un integer. Por favor, pruebe otra vez!");
          }
     }
   }

   /**
    * Lee un double desde teclado. La entrada termina con un salto de linea
    * @return el valor cargado, como un double
    */
   public static double readDouble()
   {
     while(true)
     {
          try
          {
             return Double.parseDouble(readLine().trim());
          }
          catch(NumberFormatException e)
          {
             System.out.println("No es un flotante. " + "Por favor, pruebe otra vez!");
          }
     }
   }
}
```

```
package persistencia;

/**
 * Indica el comportamiento que debe ser capaz de mostrar un objeto que vaya a
 * ser grabado en un Archivo.
 *
 * @author Ing. Valerio Frittelli
 * @version Julio de 2005
 */

import java.io.*;

public interface Grabable
{
   /**
    *  Calcula el tamaño en bytes del objeto, tal como será grabado
    *  @return el tamaño en bytes del objeto
    */
   int sizeOf();


   /**
    *  Indica cómo grabar un objeto
    *  @param el archivo donde será grabado el objeto
    */
```

```
   void grabar (RandomAccessFile a);

   /**
    *  Indica cómo leer un objeto
    *  @param a el archivo donde se hará la lectura
    */
   void leer (RandomAccessFile a);
}

// package persistencia;

/**
 * Una clase para describir en forma genérica un registro capaz de ser almacenado en un Archivo. Contiene un atributo
 * "activo" que indica si el registro es válido o no dentro del archivo. Si el registro está marcado como borrado, el
 * atributo "activo" vale false. Si el registro no está borrado, "activo" vale true.
 *
 * @author Ing. Valerio Frittelli
 * @version Julio de 2005
 */

import java.io.*;
public class Register implements Grabable
{
    private boolean activo;    // marca de registro activo. Ocupa 1 byte en disco
    private Grabable datos;    // los datos puros que serán grabados

    /**
     *  Crea un Registro sin datos, marcándolo como activo
     */

    public Register()
    {
       activo = true;
    }

    /**
     *  Crea un Registro con datos, marcándolo como activo
     */
    public Register (Grabable d)
    {
       activo = true;
       datos = d;
    }


    /**
     *  Determina si el registro es activo o no
     *  @return true si es activo, false si no.
     */
    public boolean isActive ()
    {
       return activo;
    }

    /**
     *  Cambia el estado del registro, en memoria
     *  @param x el nuevo estado
     */
```

```
public void setActive(boolean x)
{
  activo = x;
}

/**
 *  Cambia los datos del registro en memoria
 *  @param d los nuevos datos
 */
public void setData(Grabable d)
{
  datos = d;
}

/**
 *  Accede a los datos del registro en memoria
 *  @return una referencia a los datos del registro
 */
public Grabable getData()
{
   return datos;
}

/**
 *  Calcula el tamaño en bytes del objeto, tal como será grabado en disco. Pedido por Grabable.
 *  @return el tamaño en bytes del objeto como será grabado.
 */
public int sizeOf()
{
  return datos.sizeOf() + 1;   // suma 1 por el atributo "activo", que es boolean
}

/**
 *  Especifica cómo se graba un Register en disco. Pedido por Grabable.
 *  @param a el manejador del archivo de disco donde se hará la grabación
 */
public void grabar (RandomAccessFile a)
{
   try
   {
      a.writeBoolean(activo);
      datos.grabar(a);
   }
   catch(IOException e)
   {
      System.out.println("Error al grabar el estado del registro: " + e.getMessage());
      System.exit(1);
   }
}

/**
 *  Especifica cómo se lee un Register desde disco. Pedido por Grabable.
 *  @param a el manejador del archivo de disco desde donde se hará la lectura
 */
public void leer (RandomAccessFile a)
{
   try
   {
      activo = a.readBoolean();
      datos.leer(a);
```

```
        }
        catch(IOException e)
        {
           System.out.println("Error al leer el estado del registro: " + e.getMessage());
           System.exit(1);
        }
    }

    /**
     * Lee desde un archivo un String de "tam" caracteres. Se declara static para que pueda ser usado en forma global por cualquier
     * clase que requiera leer una cadena de longitud fija desde un archivo.
     * @param  arch el archivo desde el cual se lee
     * @param  tam la cantidad de caracteres a leer
     * @return el String leido
     */
    public static final String readString (RandomAccessFile arch, int tam)
    {
       String cad = "";

       try
       {
          char vector[] = new char[tam];
          for(int i = 0; i<tam; i++)
          {
            vector[i] = arch.readChar();
          }
          cad = new String(vector,0,tam);
       }
       catch(IOException e)
       {
         System.out.println("Error al leer una cadena: " + e.getMessage());
         System.exit(1);
       }

       return cad;
    }

    /**
     * Graba en un archivo un String de "tam" caracteres. Se declara static para que pueda ser usado forma en global por cualquier
     * clase que requiera grabar una cadena de longitud fija en un archivo.
     * @param  arch el archivo en el cual se graba
     * @param  cad la cadena a a grabar
     * @param  tam la cantidad de caracteres a grabar
     */
    public static final void writeString (RandomAccessFile arch, String cad, int tam)
    {
       try
       {
          int i;
          char vector[] = new char[tam];
          for(i=0; i<tam; i++)
          {
            vector[i]= ' ';
          }
          cad.getChars(0, cad.length(), vector, 0);
          for (i=0; i<tam; i++)
          {
            arch.writeChar(vector[i]);
```

```
            }
        }
        catch(IOException e)
        {
            System.out.println("Error al grabar una cadena: " + e.getMessage());
            System.exit(1);
        }
    }
}

// package persistencia;

/**
 * Una clase para representar un archivo binario, cuyo contenido son registros uniformes (del mismo tipo) y de la misma longitud.
 * El archivo no permite grabar objetos cuyo tipo y tamaño no coincida con los que se indicaron en el constructor.
 * @author  Ing. Valerio Frittelli
 * @version Julio de 2005
 */

import java.io.*;
public class Archivo
{
    private File fd;                 // descriptor del archivo para acceder a sus propiedades externas
    private RandomAccessFile maestro;  // objeto para acceder al contenido del archivo
    private Grabable tipo;             // representa al contenido grabado en el archivo
    private Register reg;              // auxiliar para operaciones internas

    /**
     * Crea un manejador para el Archivo, asociando al mismo un nombre a modo de file descriptor, y un tipo de contenido al
     * que quedará fijo. El segundo parámetro se usa para fijar el tipo de registro que será aceptado para grabar en el archivo.
     * No se crea el archivo en disco, ni se abre. Sólo se crea un descriptor general para él. La apertura y eventual creación,
     * debe hacerse con el método openForReadWrite().
     * @param nombre es el nombre físico del archivo a crear
     * @param r una instancia de la clase a la que pertenecen los objetos cuyos datos serán grabados. La instancia referida por
     *        r NO será grabada en el archivo
     * @throws ClassNotFoundException si no se informa correctamente el tipo de registro a grabar
     */
    public Archivo (String nombre, Grabable r) throws ClassNotFoundException
    {
        if (r == null) throw new ClassNotFoundException("Clase incorrecta o inexistente para el tipo de registro");

        tipo = r;
        reg = new Register(r);

        fd = new File(nombre);
    }

    /**
     *  Acceso al descriptor del archivo
     *  @return un objeto de tipo File con las propiedades de file system del archivo
     */
    public File getFileDescriptor()
    {
        return fd;
```

```
    }

    /**
     *  Acceso al manejador del archivo binario
     *  @return un objeto de tipo RandomAccessFile que permite acceder al bloque físico de datos en disco, en forma directa
     */
    public RandomAccessFile getMasterFile()
    {
        return maestro;
    }

    /**
     *  Acceso al descriptor de la clase del registro que se graba en el archivo
     *  @return una cadena con el nombre de la clase del registro usado en el archivo
     */
    public String getRegisterType()
    {
        return tipo.getClass().getName();
    }

    /**
     * Borra el Archivo del disco
     */
    public void delete()
    {
        fd.delete();
    }

    /**
     * Cambia el nombre del archivo
     * @param nuevo otro Archivo, cuyo nombre (o file descriptor) será dado al actual
     */
    public void rename(Archivo nuevo)
    {
        fd.renameTo(nuevo.fd);
    }

    /**
     *  Abre el archivo en modo de sólo lectura. El archivo en disco debe existir previamente. Queda posicionado al
     *  principio del archivo.
     */
    public void openForRead ()
    {
        try
        {
            maestro = new RandomAccessFile(fd, "r");
        }
        catch(IOException e)
        {
            System.out.println("Error de apertura archivo " + fd.getName() + ": " + e.getMessage());
            System.exit(1);
        }
    }

    /**
     *  Abre el archivo en modo de lectura y grabación. Si el archivo en disco no existía, será creado. Si existía,
         *  será posicionado al principio del archivo. Mueva el puntero de registro activo con el método seekRegister()
```

```
     *  o con seekByte().
     */
    public void openForReadWrite ()
    {
       try
       {
          maestro = new RandomAccessFile(fd, "rw");
       }
       catch(IOException e)
       {
          System.out.println("Error de apertura archivo " + fd.getName() + ": " + e.getMessage());
          System.exit(1);
       }
    }

    /**
     * Cierra el archivo
     */
    public void close()
    {
       try
       {
          maestro.close();
       }
       catch(IOException e)
       {
          System.out.println("Error al cerrar el archivo " + fd.getName() + ": " + e.getMessage());
          System.exit(1);
       }
    }

    /**
      * Ubica el puntero de registro activo en la posición del registro número i. Se supone que los registros grabados son del mismo tipo,
     * y que la longitud de los registros es uniforme.
     * @param i número relativo del registro que se quiere acceder
     */
    public void seekRegister (long i)
    {
       try
       {
          maestro.seek(i * reg.sizeOf());
       }
       catch(IOException e)
       {
          System.out.println("Error al posicionar en el registro número " + i + ": " + e.getMessage());
          System.exit(1);
       }
    }

    /**
     * Ubica el puntero de registro activo en la posición del byte número b
     * @param b número del byte que se quiere acceder, contando desde el principio del archivo
     * @throws IOException si hubo problema en el posicionamiento
     */
    public void seekByte (long b)
    {
       try
       {
          maestro.seek(b);
```

```
    }
    catch(IOException e)
    {
       System.out.println("Error al posicionar en el byte número " + b + ": " + e.getMessage());
       System.exit(1);
    }
}

/**
 * Rebobina el archivo: ubica el puntero de registro activo en la posición cero
 */
public void rewind()
{
    try
    {
       maestro.seek(0);
    }
    catch(IOException e)
    {
       System.out.println("Error al rebobinar el archivo: " + e.getMessage());
       System.exit(1);
    }
}

/**
 * Devuelve el número relativo del registro en el cual esta posicionado el archivo en este momento
 * @return el número del registro actual
 */
public long registerPos()
{
    try
    {
       return maestro.getFilePointer() / reg.sizeOf();
    }
    catch(IOException e)
    {
       System.out.println("Error al intentar devolver el número de registro: " + e.getMessage());
       System.exit(1);
    }

    return -1;
}

/**
 * Devuelve el número de byte en el cual esta posicionado el archivo en este momento
 * @return el número de byte de posicionamiento actual
 */
public long bytePos ()
{
    try
    {
       return maestro.getFilePointer();
    }
    catch(IOException e)
    {
       System.out.println("Error al intentar devolver el número de byte: " + e.getMessage());
       System.exit(1);
    }

    return -1;
```

```
}

/**
 * Posiciona el puntero de registro activo al final del archivo
 */
public void goFinal ()
{
   try
   {
      maestro.seek(maestro.length());
   }
   catch(IOException e)
   {
      System.out.println("Error al posicionar al final: " + e.getMessage());
      System.exit(1);
   }
}

/**
 * Devuelve la cantidad de registros del archivo en este momento
 * @return el número de registros del archivo
 */
public long countRegisters()
{
   try
   {
      return maestro.length() / reg.sizeOf();
   }
   catch(IOException e)
   {
      System.out.println("Error al calcular el número de registros: " + e.getMessage());
      System.exit(1);
   }

   return 0;
}

/**
 * Determina si se ha llegado al final del archivo o no
 * @return true si se llegó al final - false en caso contrario
 * @throws IOException si hubo problema en la operación
 */
public boolean eof ()
{
   try
   {
      if (maestro.getFilePointer() < maestro.length()) return false;
      else return true;
   }
   catch(IOException e)
   {
      System.out.println("Error al determinar el fin de archivo: " + e.getMessage());
      System.exit(1);
   }

   return true;
}

/**
 * Graba un registro en el archivo
```

```
     * @param r el registro a grabar
     */
    public void grabar (Register r)
    {
        if(r!=null && r.getData().getClass() == tipo.getClass())
        {
            try
            {
                r.grabar(maestro);
            }
            catch(Exception e)
            {
                System.out.println("Error al grabar el registro: " + e.getMessage());
                System.exit(1);
            }
        }
    }

    /**
     * Lee un registro del archivo
     * @return el registro leido
     * @throws IOException si hubo problema en la operación
     */
    public Register leer ()
    {
        try
        {
            reg.leer(maestro);
            return reg;
        }
        catch(Exception e)
        {
            System.out.println("Error al leer el registro: " + e.getMessage());
            System.exit(1);
        }

        return null;
    }

    /**
       * Busca un registro en el archivo. Si la clase del registro que se busca no coincide con la clase de los registros grabados
     * en el archivo, retorna -1. En general, el retorno de -1 significa que el registro no fue encontrado.
     * @param r objeto a buscar en el archivo
     * @return la posición de byte del registro en el archivo, si existe, o -1 si no existe
     */
    public long buscar (Register r)
    {
        if(r == null || tipo.getClass() != r.getData().getClass()) return -1;

        long pos = -1, actual = bytePos();
        rewind();
        while (!eof())
        {
            reg = leer();
            if (reg.getData().equals(r.getData()) && reg.isActive())
            {
                pos = bytePos() - reg.sizeOf();
                break;
            }
```

```
        }
        seekByte(actual);
        return pos;
    }


    /**
     * Agrega un registro en el archivo, controlando que no haya repetición y que la clase del nuevo registro coincida con la
     * clase indicada para el archivo al invocar al constructor. El archivo debe estar abierto en modo de grabación.
     * @param r registro a agregar
     * @return true si fue posible agregar el registro - false si no fue posible
     */
    public boolean alta (Register r)
    {
        boolean resp = false;
        long pos;

        if(r != null && tipo.getClass() == r.getData().getClass())
        {
            openForReadWrite();

            try
            {
                pos = buscar(r);
                if (pos == -1)
                {
                    goFinal();
                    grabar(r);
                    resp =  true;
                }
            }
            catch(Exception e)
            {
                System.out.println("Error al grabar el registro: " + e.getMessage());
                System.exit(1);
            }

            close();
        }

        return resp;
    }

    /**
     * Agrega un registro en el archivo, sin controlar repetición. La clase del nuevo registro debe coincidir con la
     * clase indicada para el archivo al invocar al constructor. El archivo debe estar abierto en modo de grabación.
     * @param r registro a agregar
     * @return true si fue posible agregar el registro - false si no fue posible
     */
    public boolean altaDirecta (Register r)
    {
        boolean resp = false;

        if(r != null && tipo.getClass() == r.getData().getClass())
        {
            openForReadWrite();
```

```
            try
            {
               goFinal();
               grabar(r);
               resp = true;
            }
            catch(Exception e)
            {
               System.out.println("Error al grabar el registro: " + e.getMessage());
               System.exit(1);
            }

            close();
         }

         return resp;
      }

      /**
       * Borra un registro del archivo. La clase del registro buscado debe coincidir con la clase indicada para el archivo
       * al invocar al constructor. El archivo debe estar abierto en modo de grabación. El registro se marca como borrado,
       * aunque sigue físicamente ocupando lugar en el archivo
       * @param r registro a buscar y borrar
       * @return true si fue posible borrar el registro - false si no fue posible
       */
      public boolean baja (Register r)
      {
         boolean resp = false;
         long pos;

         if(r != null && tipo.getClass() == r.getData().getClass())
         {
            openForReadWrite();

            try
            {
               pos = buscar(r);
               if (pos != -1)
               {
                   seekByte(pos);
                   reg = leer();
                   reg.setActive(false);

                   seekByte(pos);
                   grabar(reg);
                   resp = true;
               }
            }
            catch(Exception e)
            {
               System.out.println("Error al eliminar el registro: " + e.getMessage());
               System.exit(1);
            }

            close();
         }

         return resp;
```

```
   }

   /**
     * Modifica un registro en el archivo. Reemplaza el registro en una posición dada, por otro tomado como parámetro.
     * La clase del registro buscado debe coincidir con la clase indicada para el archivo al invocar al constructor. El
     * archivo debe estar abierto en modo de grabación.
     * @param r registro con los nuevos datos
     * @return true si fue posible modificar el registro - false si no fue posible
     */
   public boolean modificar (Register r)
   {
      boolean resp = false;
      long pos;

      if(r != null && tipo.getClass() == r.getData().getClass())
      {
         openForReadWrite();

         try
         {
            pos = buscar(r);
            if (pos != -1)
            {
                seekByte(pos);
                grabar(r); // graba el nuevo registro encima del anterior
                resp = true;
            }
         }
         catch(Exception e)
         {
            System.out.println("Error al modificar el registro: " + e.getMessage());
            System.exit(1);
         }

         close();
      }

      return resp;
   }


   /**
     * Elimina físicamente los registros que estuvieran marcados como borrados. El Archivo queda limpio, pero sale cerrado.
     */
   public void depurar ()
   {
      try
      {
        Archivo temp = new Archivo ("temporal.dat", tipo);
        temp.openForReadWrite();

        this.openForRead();
        while (!this.eof())
        {
              reg = this.leer();
              if (reg.isActive()) temp.grabar(reg);
        }
```

```
            this.close();
            temp.close();
            this.delete();
            temp.rename(this);
        }
        catch(ClassNotFoundException e)
        {
            System.out.println("Error de tipo de dato con el archivo temporal: " + e.getMessage());
            System.exit(1);
        }
    }
}

package datos;

/**
 * Representa un estudiante de una carrera cualquiera, que podrá ser usado dentro de un Register para grabar en un Archivo
 *
 * @author Ing. Valerio Frittelli
 * @version Julio de 2005
 */

import java.io.*;
import persistencia.*;

public class Alumno implements Grabable
{
    private int     legajo;      // 4 bytes en disco
    private String  nombre;      // una cadena, que queremos sea de 30 caracteres máximo = 60 bytes en disco
    private float   promedio;    // 4 bytes en disco

    /**
     * Constructor por defecto. Los atributos quedan con valores por default
     */
    public Alumno ()
    {
    }

    /**
     * Constructor. Inicializa cada atributo de acuerdo a los parámetros
     */
    public Alumno (int leg, String nom, float prom)
    {
        legajo   = leg;
        nombre   = nom;
        promedio = prom;
    }

    /**
     * Accede al valor del legajo
     * @return el valor del atributo legajo
     */
    public int getLegajo()
    {
      return legajo;
    }
```

```
/**
 * Accede al valor del nombre
 * @return el valor del atributo nombre
 */
public String getNombre()
{
  return nombre;
}

/**
 * Accede al valor del promedio
 * @return el valor del atributo promedio
 */
public float getPromedio()
{
  return promedio;
}

/**
 * Cambia el valor del legajo
 * @param leg el nuevo valor del atributo legajo
 */
public void setLegajo (int leg)
{
  legajo = leg;
}

/**
 * Cambia el valor del nombre
 * @param nom el nuevo valor del atributo nombre
 */
public void setNombre (String nom)
{
  nombre = nom;
}

/**
 * Cambia el valor del promedio
 * @param prom el nuevo valor del atributo promedio
 */
public void setPromedio (float prom)
{
  promedio = prom;
}

/**
 *  Calcula el tamaño en bytes del objeto, tal como será grabado. Pedido por Grabable
 *  @return el tamaño en bytes del objeto
 */
public int sizeOf()
{
    int tam = 68;  // 4 + 60 + 4. ¿Alguna duda?
    return tam;
}

/**
 *  Indica cómo grabar un objeto. Pedido por Grabable.
 *  @param el archivo donde será grabado el objeto
 */
public void grabar (RandomAccessFile a)
```

```
    {
       try
       {
           a.writeInt(legajo);
           Register.writeString (a, nombre, 30);
           a.writeFloat(promedio);
       }
       catch(IOException e)
       {
           System.out.println("Error al grabar el registro: " + e.getMessage());
           System.exit(1);
       }
    }

    /**
     *  Indica cómo leer un objeto. Pedido por Grabable.
     *  @param a el archivo donde se hará la lectura
     */
    public void leer (RandomAccessFile a)
    {
       try
       {
           legajo   = a.readInt();
           nombre   = Register.readString(a, 30).trim();
           promedio = a.readFloat();
       }
       catch(IOException e)
       {
           System.out.println("Error al leer el registro: " + e.getMessage());
           System.exit(1);
       }
    }

    /**
     *  Redefinición del heredado desde Object. Considera que dos Alumnos son iguales si sus legajos lo son
     *  @param x el objeto contra el cual se compara
     *  @return true si los legajos son iguales, false en caso contrario
     */
    public boolean equals (Object x)
    {
       if(x==null) return false;

       Alumno a = (Alumno)x;
       return (legajo == a.legajo);
    }

    /**
     *  Redefinición del heredado desde Object. La convención es si equals() dice que dos objetos son iguales,
entonces
     *  hashCode() debería retornar el mismo valor para ambos.
     *  @return el hashCode del Alumno. Se eligió el número de legajo para ese valor.
     */
    public int hashCode ()
    {
       return legajo;
    }

    /**
     *  Redefinición del heredado desde Object.
     *  @return una cadena representando el contenido del objeto.
```

```
     */
    public String toString()
    {
       return "\nLegajo: " + legajo + "\tNombre: " + nombre + "\tPromedio: " + promedio;
    }
}

package datos;

/**
 * Representa un artículo a la venta en un comercio cualquiera, que podrá ser usado dentro de un Register para grabar en un Archivo
 *
 * @author Ing. Valerio Frittelli
 * @version Julio de 2005
 */

import java.io.*;
import persistencia.*;
```

public class Articulo implements Grabable

```
{
    private int     codigo;       // 4 bytes en disco
     private String  descripcion;  // una cadena, que queremos sea de 50 caracteres máximo = 100 bytes en disco

    /**
     * Constructor por defecto. Los atributos quedan con valores por default
     */
    public Articulo ()
    {
    }

    /**
     * Constructor. Inicializa cada atributo de acuerdo a los parámetros
     */
    public Articulo (int cod, String nom)
    {
       codigo   = cod;
       descripcion = nom;
    }

    /**
     * Accede al valor del codigo
     * @return el valor del atributo codigo
     */
    public int getCodigo()
    {
      return codigo;
    }

    /**
     * Accede al valor de la descripción
     * @return el valor del atributo descripcion
     */
    public String getDescripcion()
    {
      return descripcion;
    }
```

```
/**
 * Cambia el valor del codigo
 * @param c el nuevo valor del atributo codigo
 */
public void setCodigo (int c)
{
  codigo = c;
}

/**
 * Cambia el valor de la descripcion
 * @param nom el nuevo valor del atributo descripcion
 */
public void setDescripcion (String nom)
{
  descripcion = nom;
}

/**
 *  Calcula el tamaño en bytes del objeto, tal como será grabado. Pedido por Grabable
 *  @return el tamaño en bytes del objeto
 */
public int sizeOf()
{
     int tam = 104;  // 4 + 100 ¿Alguna duda?
     return tam;
}

/**
 *  Indica cómo grabar un objeto. Pedido por Grabable.
 *  @param el archivo donde será grabado el objeto
 */
public void grabar (RandomAccessFile a)
{
   try
   {
       a.writeInt(codigo);
       Register.writeString (a, descripcion, 50);
   }
   catch(IOException e)
   {
       System.out.println("Error al grabar el registro: " + e.getMessage());
       System.exit(1);
   }
}

/**
 *  Indica cómo leer un objeto. Pedido por Grabable.
 *  @param a el archivo donde se hará la lectura
 */
public void leer (RandomAccessFile a)
{
   try
   {
       codigo  = a.readInt();
       descripcion  = Register.readString(a, 50).trim();
   }
   catch(IOException e)
   {
       System.out.println("Error al leer el registro: " + e.getMessage());
```

```
            System.exit(1);
         }
      }

      /**
       *  Redefinición del heredado desde Object. Considera que dos Articulos son iguales si sus códigos lo son
       *  @param x el objeto contra el cual se compara
       *  @return true si los códigos son iguales, false en caso contrario
       */
      public boolean equals (Object x)
      {
         if(x==null) return false;

         Articulo a = (Articulo)x;
         return (codigo == a.codigo);
      }

      /**
       *  Redefinición del heredado desde Object. La convención es si equals() dice que dos objetos son iguales, entonces
       *  hashCode() debería retornar el mismo valor para ambos.
       *  @return el hashCode del Articulo. Se eligió el código para ese valor.
       */
      public int hashCode ()
      {
         return codigo;
      }

      /**
       *  Redefinición del heredado desde Object.
       *  @return una cadena representando el contenido del objeto.
       */
      public String toString()
      {
         return "\nCodigo: " + codigo + "\tDescripcion: " + descripcion;
      }
}
```