PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TEMA 2

PROGRAMACIÓN, JAVA. PARTE II

Contenido

1. Interfaces

El concepto de interface evoluciona la idea de clase abstracta. Una interface es una clase abstracta pura, sólo contiene la signatura o firma de sus métodos sin su implementación. También puede contener la definición de datos, pero estos serán constantes. Existe la posibilidad de declarar e implementar métodos son los llamados métodos por defecto y tienen el código por defecto que se ejecutara si una clase derivada no redefine el método heredado de la interface. Otros métodos que se pueden implementar en una interface son los métodos estáticos que son propiedad de la clase y se declaran como static. Otros modificadores de acceso pueden ser aplicados en la definición de métodos dentro de una interface.

Sintaxis de una interface:

```
public interface nombreInterface {
   [static final tipo CONSTANTE = valor;]
   public tipoDevuelto nombreMetodo(lista de parámetros);
   [default tipoDevuelto nombreMetodo(lista de parámetros) {
         // cuerpo del método
   }]
   [[modificador de acceso] static tipoDevuelto nombreMetodo(lista de parámetros) {
         // cuerpo del método
         }]
}
```

Las características de la interface son las siguientes:

- Son colecciones de métodos y constantes.
- Todos los métodos son abstractos.
- El acceso a una interface es público:
  - Los métodos son public.
  - Los atributos son public, static y final.
  - Las declaraciones de acceso son implícitas, es decir no es necesario indicar el tipo de acceso.
- Son elementos totalmente de diseño.
- No pueden contener métodos constructores, no se pueden instanciar.

Nota: a partir de la versión Java 8 se añaden nuevas funcionalidades a los interfaces, lo que amplia y matiza las características de los interfaces. A continuación, se habla de los detalles de las interfaces.

1.1 Herencia

Una clase puede heredar de una interface, pero debe implementar sus métodos. La palabra clave para realizar la herencia es *implements*. Las interfaces permiten la **herencia múltiple** en las clases, una clase puede implementar múltiples interfaces. Ejemplo de la herencia de una interface:

Herencia entre interfaces

Una interface puede heredar múltiples interfaces. La palabra clave es **extends**. La sintaxis es la siguiente:

```
interface nombre_interface  extends nombre_interface  , . . . {
    tipo_retorno nombre_metodo ( lista_argumentos ) ;
    . . .
}
```

1.2 Interfaces como contenedores de constantes

Los datos que se definen en una interface son implícitamente de tipo public, static y final, es decir son constantes de acceso público. Resulta útil emplear las interfaces para definir conjuntos de constantes. El acceso a los datos de la interface se hace empleando la notación: nombreInterface.dato . Por ejemplo el número de los meses del año o su nombre.

```
public interface Meses {
    int ENERO = 1 , FEBRERO = 2 . . . ;
    String [] NOMBRES_MESES = { " " , "Enero" , "Febrero" , . . . };
}


System.out.println(Meses.NOMBRES_MESES[ENERO]);
```

1.3 Instancia de una interface

Java considera las interfaces como tipos de datos. Lo anterior implica que podemos crear variable del tipo de una interfaz. Si creamos una clase que implementa una interface, entonces podemos asignar la instancia de un objeto de dicha clase a una variable de interface. De esta forma, se puede implementar el **polimorfismo** con interfaces. Vemos un ejemplo:

```
public interface Vehiculo {
    public String matricula = "";
    public float maxVel
    public void arrancar();
    public void detener();
    default void claxon(){
      System.out.println("Sonando claxon");
   }
}
```

```
public class Coche implements Vehiculo {
    public void arrancar() {
        System.out.println("arrancando motor...");
    }
    public void detener() {
        System.out.println("deteniendo motor...");
    }
}
```

```
Vehiculo tesla = new Coche();

tesla.arrancar(); // arrancar el motor...
```

1.4 Interfaces y proliferación de nombres

Una aplicación tiene un numero de clases e interfaces. El mantenimiento del software aumenta dicha cantidad de clases e interfaces. Si dos o mas interfaces emplean la *misma signatura* de un método, la llamada al método confunde al compilador de Java y se produce un error en tiempo de compilación. La solución consiste en llamar al método del interface deseado y emplear la palabra reservada "super". A continuación se muestra un ejemplo:

```
public interface Radio {
    // public void iniciarRadio();
    // public void detenerRadio();

    default public void siguiente() {
        System.out.println("Siguiente emisora de Radio");
    }
}

public interface ReproductorMusica {
    // public void iniciar();
    // public void pausar();
    // public void detener();

    default public void siguiente() {
        System.out.println("Siguiente canción del Reproductor de Música");
    }
}

public class Smartphone implements Radio, ReproductorMusica {
    public void siguiente() {
        // Supongamos que deseas llamar a siguiente del ReproductorMusica
        ReproductorMusica.super.siguiente();
    }
}

Smartphone motoG = new Smartphone();
motoG.siguiente();  // Siguiente desde ReproductorMusica
```

1.5 ¿ Interfaces o clases abstractas ?

Utilizaremos una clase abstracta en una jerarquía de clases, cuando necesitemos definir una plantilla para un grupo de clases derivadas.

Si necesitamos definir un rol para un grupo de clases, independientemente de la jerarquía de clases que ocupen, emplearemos una interface.

2. Entrada y salida

Java, como todos los lenguajes de programación, tiene mecanismos para llevar a cabo operaciones de entrada y salida de datos. Los datos son de entrada cuando son recibidos por la aplicación desde un origen de datos, denominado generalmente fuente de datos. Los datos son de salida cuando la aplicación envía datos hacia un destino de datos, denominado generalmente sumidero de datos.

Las operaciones de entrada y salida de datos son sensibles a los errores y por ello suelen generar excepciones. El uso de operaciones de entrada y salida conlleva la gestión de excepciones en la aplicación.

Java utiliza un concepto de entrada/salida transversal, lo que significa que es independiente del dispositivo con el que se trabaja. El mecanismo empleado por Java son los Flujos(Streams).

2.1 Flujos

Un flujo es un canal por donde circula la información y permite tratar la información entre el programa y el exterior. Los flujos pueden ser unidireccionales (entrada, salida) o bidireccionales (entrada/salida). Los flujos son independientes del dispositivo al que estén conectados.

Java tiene dos grandes categorías de flujos con sus respectivas clases, los flujos de bytes y los flujos de caracteres. El empleo de un grupo concreto depende de la naturaleza de los datos.

2.2 Ficheros

Hasta ahora cada vez que hemos ejecutado un programa éste se ha ejecutado desde cero. Pero si quisiéremos poder ejecutar un programa recuperando los valores de la última vez que se ha ejecutado necesitaríamos almacenar estos valores en estructuras de datos persistentes como un fichero o una base de datos.

Un fichero o archivo es una secuencia de bytes almacenada en la memoria persistente de la computadora e identificada y gestionada por un sistema de ficheros formando una entidad lógica.

Cada fichero tiene asociado un nombre que ha de ser único dentro del directorio en el que se crea. Eso convierte la ruta hacia el directorio en el identificador inequívoco del fichero.

El tamaño y contenido de un fichero puede variar en cualquier momento. El sistema de ficheros del sistema operativo será el encargado de controlar en todo momento quien tiene acceso al fichero evitando que dos programas lo puedan modificar en el mismo instante. Cada vez que queramos modificar o leer un fichero tendremos que pedir permiso al sistema de ficheros.

En función de cómo queramos acceder a un fichero para leer o modificarlo vamos a poder diferenciar dos métodos de acceso:

- Acceso secuencial: accedemos a los valores contenidos en el fichero como si fuera un archivo de texto, empezando por la primera palabra hasta que encontremos el valor que queremos y decidamos detener la lectura. Es un método cuya rapidez dependerá de lo pronto o tarde que encuentre la información.
- Acceso directo: accedemos directamente a la posición del contenido que quiero modificar. Este acceso es más rápido pero implica conocer exactamente la posición en donde quiero leer/modificar un dato.

Aunque podríamos modificar manualmente cada 0 y 1 que componen el contenido de un fichero, utilizaremos funcionalidades Java para leer y escribir texto formateado.

2.3 La clase File de Java. Gestión de archivos y carpetas

Para trabajar con ficheros utilizaremos variables del tipo File. Estas variables nos van a permitir gestionar toda la información relativa a ficheros y carpetas a partir de la ruta de acceso al archivo o carpeta. Las funciones de File relativas al tratamiento de carpetas no funcionarán si la variable hace referencia a un archivo y evidentemente las funciones relativas al tratamiento de archivos no funcionarán si la variable File hace referencia a una carpeta.

Para crear una nueva variable del tipo File utilizaremos la sintaxis: File nombreVariableFile = new File ("ruta al archivo o carpeta");

Para utilizar alguna de sus funciones asociadas utilizaremos la sintaxis: nombreVariableFile.nombreFuncion();

Algunas de las funciones más importantes que va a tener una variable File son:

- .isDirectory(); Retorna true o false según la variable haga referencia a un directorio o no.
- .isFile(); Retorna true o false según la variable haga referencia a un archivo o no.
- .exists(); retorna true o false según haga referencia a un archivo o directorio que exista o no.
- .canRead(); retorna true o false según tengamos o no permisos de lectura.
- .canWrite(); retorna true o false según tengamos o no permisos de escritura.
- .length(); retorna un valor long con la longitud del archivo.
- .getName(); retorna una String el nombre del fichero o carpeta.
- .getParent() retorna el nombre de la carpeta padre que contiene el fichero o carpeta.
- .getPath(): retorna la ruta relativa del archivo.
- .mkdir(); crea una carpeta en la ruta indicada en la variable File siempre que no exista previamente.

Retorna true si se ha podido crear y false en caso contrario.

- .createNewFile(); crea un archivo en blanco en la ruta definida en la variable File siempre que no exista previamente. Retorna true o false según si se ha podido crear o no.
- .renameTo(File); la variable File desde la que se llama la función se le asigna el nombre de la variable File pasada como parámetro. Las dos variables han de apuntar a la misma carpeta. Retorna true o false según si se ha podido renombrar o no.
- .delete(); borra el archivo o carpeta referenciados en la variable File. Retorna true o false según se haya o no podido eliminar el archivo o carpeta.
- .list(); retorna un array de Strings con el nombre de los ficheros contenidos en la carpeta referenciada en la variable File. Es posible pasar un filtro como parámetro para obtener solo un tipo de ficheros.
- .listFiles(); retorna un array de variables del tipo File que hacen referencia a los ficheros y directorios contenidos dentro de la carpeta referenciada en la variable File.

2.4 Elementos básicos en la lectura y escritura de información

Java clasifica la escritura y lectura de datos entre aquellas operaciones que se dan entre las propias variables del programa y aquellas operaciones con elementos externos al programa, los que llamaremos periféricos. Un periférico va a poder ser el teclado, la pantalla, un fichero, una impresora, un scanner, … y siempre que trabajemos leyendo o escribiendo datos contra un periférico utilizaremos dos grandes grupos de variables: los Streams y los Buffers.

Cuando creemos una variable del tipo Stream le vamos a tener que vincular un periférico. A través de esta variable podremos escribir o leer bytes con el periférico.

Como nosotros no estamos acostumbrados a trabajar con bytes, a cada Stream que creemos le asociaremos un Buffer que nos permitirá comunicarnos con el Stream a través de Strings en vez de bytes.

Ya veremos que de variables tipo Stream y Buffer encontraremos una gran variedad dependiendo del tipo de periférico con el que queramos conectar, del tipo de acceso y si vamos a realizar operaciones de lectura y escritura.

La gran mayoría de variables del tipo Stream y Buffers las encontraremos en el package: java.io . El nombre java.io hace referencia a operaciones de Input/Output (lectura/Escritura). Así que deberemos importar la librería con: import java.io.*;

2.5 Ficheros. Escritura secuencial

Para la escritura secuencial sobre un fichero utilizaremos el siguiente tipo de variables:

- File : para gestionar la información del fichero. Al crear la variable del tipo File le vincularemos la ruta al fichero con la sintaxis:
  - File variableFile = new File(ruta_al_documento);
- FileWriter: como variable del tipo Stream para conectar y escribir sobre el fichero. Al crear la variable le vincularemos el fichero con la sintaxis: FileWriter variableFileWriter = new FileWriter(variableFile); FileWriter también admite como parámetro la ruta al documento:
  - FileWriter variableFileWriter = new FileWriter (ruta_al_documento);
- BufferedWriter: como variable del tipo Buffer a la que escribiremos Strings para que se los pase a FileWriter. Al crear la variable le vincularemos el stream FileWritter con la sintaxis:
  - BufferedWriter variableBuffer = new BufferedWriter(variableFileWriter);

Una vez creadas las variables necesarias utilizaremos las funciones de la variableBuffer para escribir, vaciar y cerrar la comunicación con el fichero. La sintaxis será la siguiente:

- variableBuffer.write("texto a escribir"); : para escribir un texto en el fichero.
- variableBuffer.flush(); : para escribir todo aquello que pudiera haber quedado pendiente en el buffer y vaciarlo.
- variableBuffer.close(); : para finalizar la comunicación con el fichero.

En el siguiente código se muestra un ejemplo en el que se accede a un documento con la ruta "d:\ejemplo.txt" para escribir el texto "Nuevo texto" y finalmente cerrar la conexión:

```
import java.io.*;

public class EjemploEscrituraFichero {

   public static void main(String[] args) throws IOException {
      String ruta_al_documento="d:/ejemplo.txt";
      File document = new File(ruta_al_documento);  //información sobre el fichero
      FileWriter fw = new FileWriter(document);  //Stream conectado al fichero a escribir.
      BufferedWriter bw = new BufferedWriter(fw);  //Buffer que almacena datos hacia el stream
      bw.write("Nuevo texto"); //guarda los datos en el buffer
      bw.flush();  //envia los datos que queden al buffer
      bw.close();  //se liberan los recursos asignados al outputStream
   }
}
```

Nota: cada vez que abramos el archivo para escribir en él se borrarán todos los datos contenidos. Si queremos añadir contenido tendremos que leer el contenido, guardarlo en una String, modificar la String y escribirla como el contenido del nuevo archivo.

2.6 Ficheros. Lectura secuencial

Para la lectura secuencial de un archivo utilizaremos el siguiente tipo de variables:

- File : para gestionar la información del archivo. Al crear la variable del tipo File le vincularemos la ruta al archivo con la sintaxis:
  - File variableFile = new File(ruta_al_documento);
- FileReader: será el Stream que conectará con archivo para poder leer su contenido byte a byte. Al crear la variable le vincularemos el archivo con la sintaxis: FileReader variableFileReader = new FileReader(variableFile); FileReader también admite como parámetro la ruta al documento:
  - FileReader variableFileReader = new FileReader(ruta_al_documento);
- BufferedReader: lo vincularemos con FileReader y nos va a permitir leer el contenido del archivo línea a línea. Al crear la variable le vincularemos el stream FileReader con la sintaxis:
  - BufferedReader variableBuffer = new BufferedReader (variableFileReader);

Una vez creadas las variables necesarias utilizaremos las funciones de la variableBuffer para leer una línea y cuando queramos cerrar la comunicación con el archivo. La sintaxis será la siguiente:

- variableBuffer.readLine(); : para leer una línea del archivo (hasta encontrar un salo de línea).
- variableBuffer.close(); : para finalizar la comunicación con el archivo.

En el siguiente código se muestra un ejemplo en el que se accede a un documento con la ruta "d:\ejemplo.txt" para escribir el texto "Nuevo texto" y finalmente cerrar la conexión:

```
import java.io.*;

public class EjemploEscrituraFichero {

   public static void main(String[] args) throws IOException {
      String ruta_al_documento="d:/ejemplo.txt";
      File document = new File(ruta_al_documento); //información sobre el fichero

      FileReader fr = new FileReader(document); //Stream conectado al fichero a leer.
      BufferedReader br = new BufferedReader(fr); //Buffer que almacena datos del stream
      String linea = br.readLine(); //leemos una línea del documento
      br.close(); //cerramos el buffer

      System.out.println(linea);  //mostramos por consola el texto leído   }
}
```

Nota: cada vez que ejecutemos br.readLine() obtendremos una string con el siguiente contenido almacenado en el archivo hasta encontrar un salto de línea. Si mi archivo contiene 3 líneas cada vez que ejecute br.readLine() me retornará una línea, la cuarta vez me retornará un null porque ya no hay más texto dentro del archivo.

Teniendo presente que obtendremos un null cuando ya no queden más datos en un archivo, podemos crear un bucle para mostrar todo el contenido de un archivo ejecutando en cada iteración un br.readLine() hasta obtener null;

El siguiente ejemplo muestra una posible solución para mostrar todo el contenido de un array utilizando un bucle for.

```
for( String linea=br.readLine(); linea != null; linea=br.readLine() ){
        System.out.println(linea);  //mostramos por consola el texto leído
}
```

2.7 Rutas, separadores de rutas y saltos de línea

Las direcciones en las que se indica todo el path completo las llamaremos "**rutas absolutas**" y las evitaremos utilizar a toda costa en la programación. En vez de ello utilizaremos "**rutas relativas**", que serán rutas en donde como programadores solo indicaremos la ubicación del archivo respecto a nuestro programa y la ubicación del programa se lo preguntaremos al Sistema Operativo.

Para conocer la ruta absoluta a nuestro proyecto utilizaremos la función getProperty("user.dir"); de System.

La siguiente línea guarda en rutaProyecto la ruta hasta nuestro proyecto:

String rutaProyecto = System.getProperty("user.dir");

Para conocer qué carácter utiliza el sistema operativo que ejecuta mi programa para **separar las carpetas** en una ruta utilizaremos la propiedad separator de File.

La siguiente línea guarda en separador el carácter utilizado para separar carpetas:

String separador = File.separator

Para conocer qué String utiliza el sistema operativo que ejecuta mi programa para indicar un **salto de línea** separator utilizaremos la función getProperty("line.separator"); de System.

La siguiente línea guarda en saltoLinea la string utilizada para insertar un salto de línea:

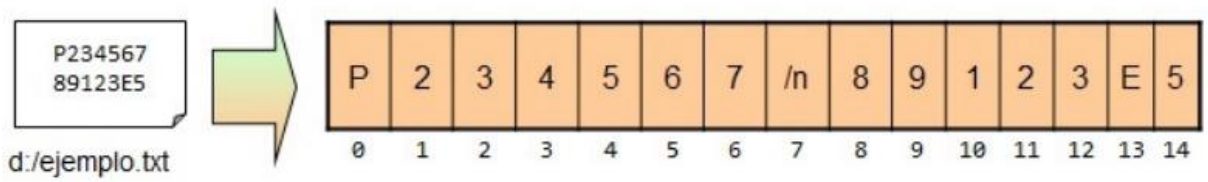
String saltoLinea = System.getProperty("line.separator");

2.8 Ficheros. Acceso directo

Hasta ahora hemos visto como acceder a ficheros de forma secuencial. Al acceder directamente a un fichero su contenido va a ser tratado como un array de chars al que tendremos asociado una variable del tipo RandomAccessFile que contendrá una variable apuntando a una posición de ese array. Cada vez que queramos leer o escribir en el archivo deberemos apuntar a una posición en concreto y luego o bien leer o escribir un carácter.

Cuando nos refiramos al "puntero" de RandomAccessFile nos estaremos refiriendo a esa variable que va a apuntar a una dirección en concreto.

El siguiente esquema ilustra cómo un fichero de texto es transformado en un array de bytes (cada byte contiene una variable char).



En Java vamos a poder acceder a de forma directa al contenido de un fichero utilizando una variable del tipo RadomAccessFile. La sintaxis va a ser:

RandomAccessFile variable = new RandomAccessFile("ruta archivo","modoAcceso");

El modo de acceso es una string que puede tener dos valores:

- "r" indicando que queremos acceder al archivo solo para leer su contenido.
- "rw" indicando que queremos acceder al archivo para leer y modificar su contenido.

Las principales funciones que RandomAccessFile nos ofrece para trabajar con archivos son las siguientes:

- .seek(numero); sitúa el puntero en la posición indicada por el número. La primera posición es la 0.
- .readByte(); retorna el byte en el que se ha situado el puntero.
- writeBytes("texto"); se sustituyen los bytes siguientes a la posición en donde esté el punto por cada carácter del texto pasado como String. Automáticamente se incrementa la longitud del archivo si es necesario.
- .readLine(); avanza el puntero hasta el primer salto de línea o fin de fichero y retorna una string con todos los caracteres leídos.
- .length(); retorna la longitud del fichero.
- .getFilePointer(); retorna la posición actual del puntero.

```
String sep = File.separator;
String rutaProyecto = System.getProperty("user.dir");
String saltoLinea = System.getProperty("line.separator");

String rutaDocumento = rutaProyecto + sep + "ejemplo.txt";
RandomAccessFile raf = new RandomAccessFile(rutaDocumento, "rw");
String text = "primera linea" +saltoLinea+ "segunda línea";
raf.writeBytes(text);    //escribir texto al inicio del archivo

raf.seek(raf.length()); //añadir informacion al final del archivo
raf.writeBytes(saltoLinea+" nueva línea añadida");
    //como leer todo un archivo
raf.seek(0);   //mover el puntero a la posición 0 del archivo
while (raf.getFilePointer() < raf.length()) { //recorremos todo el archivo línea a línea
   System.out.println(raf.readLine());     //Leer una línea
}
raf.close(); //Tancamos el fichero
```

2.9 Serialización

La serialización es un proceso mediante el cual un objeto es transformado en una secuencia de bytes que representa el estado de dicho objeto, es decir, el valor de sus atributos, para luego ser guardado en un fichero, ser enviado por la red, etc. Un objeto serializado se puede recomponer luego sin ningún problema.

Para hacer que una clase pueda ser serializable hay que hacer que implemente la interfaz Serializable. Esta interfaz no define ningún método, por lo que no tendremos que implementar nada nuevo en nuestra clase.

```
import java.io.Serializable;
public class Empleado implements Serializable{
   private String nombre;
   private int edad;
   private double sueldo;

   public Empleado(String nombre, int edad, double sueldo) {
         super();
         this.nombre = nombre;
         this.edad = edad;
         this.sueldo = sueldo;
   }
   public String getNombre() {
         return nombre;
   }
   public void setNombre(String nombre) {
         this.nombre = nombre;
   }
   public int getEdad() {
         return edad;
   }
   public void setEdad(int edad) {
         this.edad = edad;
   }
   public double getSueldo() {
         return sueldo;
   }
   public void setSueldo(double sueldo) {
         this.sueldo = sueldo;
   }
   @Override
   public String toString() {
         return "Empleado [nombre=" + nombre + ", edad=" + edad + ", sueldo=" + sueldo + "]";
   }
}
```

Una vez la clase se ha marcado como serializable, ya podemos guardar los objetos de esta clase en un fichero como una secuencia de bytes. Java se encarga de este proceso.

2.9.1 Clase ObjectOutputStream y clase ObjectInputStream

La clase ObjectOutputStream es la que usaremos para guardar objetos en un fichero. Por su parte, la clase ObjectInputStream será la encargada de recuperar los datos de los objetos del fichero.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Principal {
    public static void main(String[] args) throws FileNotFoundException, IOException,
    ClassNotFoundException {
            Empleado e1 = new Empleado("e1", 44, 30000);
            File archivo = new File("Empleados");
            //ESCRIBIR EN EL ARCHIVO
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(archivo));
            oos.writeObject(e1);
            oos.close();
            //LEER EL ARCHIVO
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream(archivo));
            Empleado e2 = (Empleado) ois.readObject();
            ois.close();
            System.out.println(e2);
    }
}
```

En el código anterior vemos que creamos un objeto de la clase Empleado que se ha implementado antes y que se ha definido como serializable. Es importante destacar este hecho ya que, si intentásemos guardar en un fichero un objeto que no se haya definido como serializable, obtendríamos un error de compilación.

Para guardar el objeto en el fichero "Empleados" se crea un objeto de la clase ObjectOutputStream sobre el fichero, y se llama al método writeObject, pasándole como parámetro el objeto a guardar.

Para leer el objeto del fichero, se crea un objeto de la clase ObjectInputStream, y se obtienen los datos del empleado con el método readObject. Para realizar la asignación se hace un casting indicando entre paréntesis que los datos leídos deben convertirse en un objeto de tipo Empleado.

No hay que olvidarse de cerrar los flujos de salida y entrada de datos una vez ya hemos escrito o leído todo lo que queríamos. Para ello usamos el método close que está implementado tanto en la clase ObjectOutputStream como en la clase ObjectInputStream.

2.9.2 El modificador transient

En algunas ocasiones querremos que uno o varios atributos de una clase serializable no se incluyan en la secuencia de bytes que representa el estado del objeto. Es decir, no queremos que se guarde el valor de un atributo. Pensemos, por ejemplo, en el caso de guardar los datos de los usuarios en un fichero. ¿Debemos guardar también sus contraseñas?

Para que el valor de un atributo no se guarde cuando serializamos un objeto usaremos el modificador transient.

```
import java.io.Serializable;
public class Empleado implements Serializable{
   private String nombre;
   private int edad;
   private double sueldo;
   private transient String clave = "12345";
   public Empleado(String nombre, int edad, double sueldo) {
         super();
         this.nombre = nombre;
         this.edad = edad;
         this.sueldo = sueldo;
   }
   ...
}
```

En el código anterior vemos el atributo clave de la clase Empleado declarado como transient. Esto hace que, en caso de que un empleado se serializase, no se guardaría el valor del atributo clave.

2.10 Entrada desde teclado

A través de la técnica empleada con los flujos se consigue realizar la entrada por teclado. Vemos una forma de realizar la entrada de datos por teclado:

```
1 InputStream is = System.in; // El teclado es Java es System.in
2 InputStreamReader isr = new InputStreamReader(is); // Lo decoramos como
      un flujo de caracteres
3 BufferedReader br = new BufferedReader(isr); // Lo decoramos con un
      flujo con memoria intermedia
4 String linea = br.readLine(); // Ya podemos leer cadenas de texto desde
      el teclado.
```

1. definimos la referencia "is" hacia el teclado, que es el flujo de entrada de byte desde el que queremos leer.
2. convertimos el flujo de entrada bytes a un flujo de entrada de caracteres con la ayuda de la clase InputStreamReader, en este momento ya podríamos leer caracteres desde el teclado, pero es más eficiente utilizar una memoria intermedia.
3. estamos creando una instancia de la clase BufferedReader sobre el flujo de entrada de caracteres (InputStreamReader).
4. leer cadenas de caracteres con la ayuda del método String readLine().

Otra forma de realizar la entrada de datos por teclado es la siguiente:

```
1 Scanner lectorTeclado = new Scanner(System.in);
2 System.out.print("Introduce un entero: ");
3 int entero = lectorTeclado.nextInt();
4 System.out.println("Introduce un real: ");
5 float real = lectorTeclado.nextFloat();
6 System.out.println("Enero = " + entero + "; real = " + real);
```

1 → declaramos la variable de tipo Scanner(Scanner lectorTeclado) y recibimos el flujo de datos de entrada desde teclado (System.in).

3 → leemos un valor entero desde teclado.

5 → leemos un valor real desde teclado.

3. Excepciones

Durante la ejecución de nuestros programas se pueden producir diversos errores, que deberemos controlar para evitar fallos. Algunos ejemplos de estos errores en tiempo de ejecución son:

- El usuario introduce datos erróneos (letras cuando se esperan números).
- Se intenta realizar una división entre cero.
- Se quiere abrir un archivo que no existe.

Todas estas situaciones se pueden controlar para que el programa no se cierre de manera inesperada, y actúe de la forma que nosotros le digamos.

Los errores antes comentados son lo que se conoce como excepciones en Java.

Java clasifica las excepciones en dos grupos:

- Error: producidos por el sistema y de difícil tratamiento.
- Exception: errores que se pueden llegar a dar por diversas situaciones, y que se pueden controlar y solucionar (formato numérico incorrecto, ausencia de fichero, error matemático, etc.).

3.1 Captura de excepciones

En nuestros programas debemos identificar situaciones en que se puedan producir errores, con el fin de desviar el flujo de ejecución y realizar las operaciones necesarias para solucionar el problema.

Esto se consigue con los bloques try/catch. La sintaxis es la siguiente:

```
try{
        /*Instrucciones que pueden producir un error*/
}
catch(TipoExcepcion e){
        /*Instrucciones a ejecutar si se da algún error en alguna de
        las instrucciones que hay en try de tipo TipoExcepcion*/
}
finally{
        /*Instrucciones que se ejecutarán siempre*/
}
```

En el momento en que se produjera un error de tipo TipoExcepcion en alguna de las instrucciones del bloque try, pasaría a ejecutarse el bloque catch.

Si no hay error en las instrucciones del bloque try, las del bloque catch no se ejecutan.

Por cada bloque try podemos poner varios bloques catch asociados. Así podremos dar diferente trato a ciertos tipos de excepciones, ejecutando código distinto. También podemos poner un único catch que capture todos los tipos de excepciones, usando la clase Exception, que es la clase de la que heredan el resto de las excepciones.

En caso de poner varios bloques catch, el que capture la clase Exception deberá ir en último lugar, ya que, de otra forma, no se llegarían a ejecutar nunca las excepciones tratadas en los bloques catch siguientes.

Al final de los bloques catch podemos poner opcionalmente un bloque finally. En él incluimos instrucciones que queremos que se ejecuten siempre, independientemente de si se ha producido un error o no.

Veamos un programa de ejemplo que usa el bloque try-catch vista antes:

```
import java.io.*;
public class Excepciones {
    public static void main(String[] args) {
        int dividendo, divisor, result;
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        try{
            System.out.println("Introduce el dividendo de la división: ");
            dividendo = Integer.parseInt(br.readLine());
            System.out.println("Introduce el divisor de la división: ");
            divisor = Integer.parseInt(br.readLine());
            result = dividendo/divisor;
            System.out.println("La división" + dividendo + " entre " + divisor + " da " + result);
        }
        catch(ArithmeticException e){
            System.out.println("No se puede dividir entre 0");
        }
        catch(NumberFormatException e){
            System.out.println("Has insertado letras en vez de números");
        }
        catch(Exception e){
            System.out.println("Se ha producido un error inesperado.");
        }
        finally{
            System.out.println("Gracias por utilizar este programa.");
        }
    }
```

Es importante poner el bloque catch correspondiente al tipo general Exception en último lugar. Así nos aseguramos de que se ejecutarán los bloques anteriores en caso de error (siempre que el tipo de error corresponda con el de algún bloque catch).

Otra forma de capturar excepciones es poner las palabras throws Exception en la cabecera de un método (en el main por ejemplo). De esta forma no necesitamos poner los bloques try-catch, pero cuando se produzca un error éste no será capturado, y no podremos tratarlo como deseemos.

Un método puede lanzar más de una excepción:

```
public static void main(String args[]) throws IOException, NumberFormatException, ...
```

3.2 Excepciones de usuario

Java nos proporciona una estructura de clases para tratar diversos tipos de excepciones. Exception es la superclase que gestiona cualquier tipo de excepción. De ella heredarán sus métodos un gran número de subclases encargadas en la gestión de excepciones específicas.

El programador puede crear sus propias clases de excepciones para tratar errores específicos que Java no contempla. Para ello hay que usar el concepto de herencia sobre la clase Exception. Normalmente, la nueva clase contendrá un único constructor con un parámetro de tipo String, que será el mensaje de error que queremos mostrar. Este mensaje lo mostraremos con el método getMessage() de la clase Exception. Veamos un ejemplo de programa que crea y lanza excepciones propias:

```
public class MiExcepcion extends Exception{
    public MiExcepcion (String mensaje){
            super(mensaje);
    }
}
```

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
            int nota;
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            try{
                System.out.println("Introduce una nota entre 0 y 10:");
                nota = Integer.parseInt(br.readLine());
                if(nota<0 || nota>10){
                        throw new MiExcepcion("Nota fuera de rango");
                }
            }
            catch(MiExcepcion e){
                System.out.println(e.getMessage());
            }
            catch(Exception e){
                System.out.println(e.getMessage());
            }
    }
}
```

3.3 Propagación de excepciones

Cuando se produce una excepción dentro de un método, se puede capturar dentro de éste y tratarlo. De no hacerse, la excepción se propaga automáticamente al método que lo llamó, y así sucesivamente hasta que un método captura la excepción o llegamos al main.

NOTA: los errores de la clase Exception no se propagan.

Observemos el siguiente ejemplo. ¿Qué ocurre cuando se produce un error de tipo NumberFormatException y no lo capturamos en el método leerNumero? Saldrá el mensaje de error que tenemos en el bloque catch en el main.

```
public static void main(String[] args) {
    int num;
    try{
            System.out.println("Introduce un número: ");
            num = leerNumero();
            System.out.println("El número es " + num);
    }
    catch(NumberFormatException e){
            System.out.println("Has insertado letras en vez de números");
    }
}
static int leerNumero(){
    int numero = 0;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    try{
            numero = Integer.parseInt(br.readLine());
    }
    catch(IOException e){
            System.out.println("ERROR");
    }
    return numero;
}
```

¿Qué ocurrirá ahora si capturamos la excepción en el método leerNumero? En este caso no debe ejecutarse el bloque catch del "main", por tanto, se ejecutará todo el código del try, incluido el System.out.println. Se mostrará un mensaje cuando no debería hacerse.

```
public static void main(String[] args) {
    int num;
    try{
            System.out.println("Introduce un número: ");
            num = leerNumero();
            System.out.println("El número es " + num);
    }
    catch(NumberFormatException e){
            System.out.println("Has insertado letras en vez de números");
    }
}
static int leerNumero(){
    int numero = 0;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    try{
            numero = Integer.parseInt(br.readLine());
    }
    catch(Exception e){
            //CAPTURAMOS EN ESTA FUNCIÓN CUALQUIER TIPO DE EXCEPCIÓN
            System.out.println("ERROR");
    }
    return numero;
}
```

Con estos ejemplos vemos la utilidad de no capturar el error dentro del método donde se produce, sino dejarlo que se propague, debido a la necesidad de tener que controlarlo fuera.

4. Colecciones

En los capítulos anteriores hemos ido haciendo programas basados en

Hasta ahora nuestros programas constaban de variables cuyos tipos de datos nos permitían guardar un único valor alfanumérico. Disponemos también de los arrays y las cadenas de caracteres (String) para poder almacenar varios valores de un mismo tipo. Incluso los objetos nos ofrecen una forma de almacenamiento de información, mediante los atributos.

Estos no son los únicos mecanismos que nos brinda Java para poder almacenar información. Hay otras estructuras avanzadas, como pueden ser las listas, colas, pilas, árboles, etc., cada una de ellas con sus métodos de acceso a los datos. En este capítulo nos centraremos en el funcionamiento de las listas (ArrayList) y colecciones (HashMap).

4.1 ArrayList

Un ArrayList lo podemos comparar con un array estático de los que se han visto en temas anteriores, por el hecho de permitirnos guardar varios valores de un mismo tipo bajo un único nombre de variable. Pero hay una diferencia muy importante entre los array y los ArrayList: éste último es dinámico. Esto quiere decir que su tamaño puede cambiar a lo largo de la ejecución del programa.

Si recordamos los arrays, en su declaración debíamos especificar el tamaño que tendrá, y este no cambiará en todo el programa. Esto puede suponer un gasto innecesario de memoria, o puede ocurrir que nos quedemos cortos al querer añadir un nuevo elemento al array. Con los arrays dinámicos, los ArraysList, este problema desaparece, ya que su tamaño se adaptará a su contenido. Durante la ejecución del programa podemos añadir y eliminar tantos elementos como queramos.

Para usar ArrayList en nuestro programa, tendremos que importar la librería java.util.ArrayList. A continuación, vemos la sintaxis de características destacables empleadas con los ArrayList.

Creación de un ArrayList vacío

```
ArrayList miLista = new ArrayList();
```

Nota: el ArrayList así creado puede contener elementos de cualquier tipo.

Añadir elementos al ArrayList

```
miLista.add("Hola");
miLista.add(31);
miLista.add('d');
miLista.add(5.5);
```

Nota: los elementos se añaden al final de la colección. El ArrayList inserta el primer elemento, que es de tipo String. Los elementos posteriores serán insertados como objetos de las clases contenedoras de los tipos básicos.

Ejercicio sobre la inserción.

Parametrización del ArrayList

```
ArrayList<tipo> miLista = new ArrayList<tipo>();
```

Nota "tipo" se refiere al tipo de datos que va a contener el ArrayList. No se pueden insertar tipos de datos básicos, se deben insertar dentro de sus clases contenedoras.

```
ArrayList<Integer> miLista = new ArrayList<Integer>();
```

Es posible insertar un elemento en un lugar concreto del ArrayList, una posición. Empleamos la función add, con otros parámetros.

```
miLista.add(0, 1000);
```

Nota: La instrucción anterior añadirá el entero 1000 en la primera posición del ArrayList (posición 0). Hay que decir que la

posición que especifiquemos debe estar comprendida entre 0 y ArrayList.size()-1, sino se producirá una excepción.

Métodos de la clase ArrayList

size()	Devuelve un entero conel número de elementos del ArrayList.
add(X)	Añade el objeto X al final del ArrayList.
add(posición, X)	Inserta el objeto X en la posición indicada del ArrayList.
get(posición)	Devuelve el elemento que está en la posición indicada.
remove(posición)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
remove(X)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos del ArrayList.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X)	Devuelve la posición del objeto X. Si no existe devuelve -1.

4.2 Metodos equals y hashCode

Contrato equals-hashCode:

"Cuando sobreescribimos el método equals() en nuestra clase para cambiar la lógica que se hereda de la clase Object, tenemos que sobreescribir el método hashCode() de manera que si dos instancias de nuestra clase son iguales según la nueva lógica en equals(), el método hashCode() deberá retornar el mismo valor si lo llamo para dichas instancias. Si la lógica de equals() dice que las instancias son distintas el

método hashCode() puede retornar cualquier valor al ser llamado para cada instancia. Es decir puede retornar el mismo valor o no".

Estos dos métodos son los que debemos usar si queremos tener control sobre cuándo dos elementos se consideran iguales. Ambos métodos son proporcionados por la interfaz Map, los cuales tendremos que sobrescribir para programar el comportamiento que queremos que tengan.

El método equals nos devuelve un booleano que indica si dos elementos son iguales o no. Por defecto, se considera que dos elementos son iguales si tienen la misma clave. Podemos ir más allá y hacer que, por ejemplo, dos elementos se consideren iguales si tienen el mismo valor. Por ejemplo, consideramos dos personas iguales si tienen el mismo nombre y apellidos.

El método hashCode viene a hacer lo mismo que el método equals, es decir, compara dos elementos y nos dice si son iguales. Pero la comparación la hace más rápida, ya que utiliza un número entero. Dos objetos que sean iguales devuelven el mismo valor hash.

Cuando comparamos dos objetos en estructuras de tipo hash (HashMap, HashSet, etc.) primero se invoca al método hashCode y luego al equals. Si los métodos hashCode de cada objeto devuelve un entero diferente, se consideran los objetos distintos. En caso contrario, en que ambos objetos tengan el mismo código hash, Java ejecutará el método equals y revisará en detalle si se cumple la igualdad.

Veamos un ejemplo en el que creamos la clase Persona e implementamos los métodos hashCode y equals:

```
public class Persona {
        String nombre;
        String apellido;
        int edad;
        public Persona(String nombre, String apellido, int edad) {
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }
        //Getters y setters
        …
}
```

Vemos que tenemos los atributos nombre, apellido y edad. Vamos a considerar que dos personas son iguales si tienen el mismo valor en los atributos nombre y apellidos.

Implementamos el método hashCode:

```
@Override
public int hashCode() {
        int result = this.nombre.hashCode() + this.apellido.hashCode();
        return result;
}
```

Lo que hacemos es declarar un entero, cuyo valor vendrá de sumar el número hash del nombre y del apellido. Se puede ver que invocamos el método hashCode sobre los Strings nombre y apellidos. La clase String tiene el método hashCode implementado, como todas las clases de Java (lo heredan de la clase Object). Lo que nosotros podemos hacer es sobreescribirlo, como estamos haciendo en la clase Persona.

El método equals considerará los mismos criterios que hashCode, es decir, dos personas son iguales si tienen el mismo nombre y apellido.

```
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	Persona other = (Persona) obj;
	if (apellido == null) {
		if (other.apellido != null)
			return false;
	} else if (!apellido.equals(other.apellido))
		return false;
	if (nombre == null) {
		if (other.nombre != null)
			return false;
	} else if (!nombre.equals(other.nombre))
		return false;
	return true;
}
```

5. Interfaces de usuario

En los capítulos anteriores hemos ido haciendo programas basados en la consola. Esto quiere decir que no usan interfaz gráfica de usuario (GUI). Pese a que los programas hechos en la consola son completamente operativos, la mayoría de las aplicaciones tienen una interfaz gráfica. En este capítulo, veremos un kit de herramientas GUI que ofrece Java, llamado Swing.

Cabe destacar que Swing ofrece una amplia variedad de componentes, por lo que en este capítulo se tratará el tema de forma superficial. Será un punto de partida para entender Swing y saber utilizarlo, siendo capaces de crear aplicaciones con una interfaz gráfica completa y funcional.

5.1 Componentes

Botón

JButton	Botón
JLabel	Etiqueta de texto
JTextField	Cuadro de texto
JCheckBox	Casilla de verificación
JRadioButton	Botón de opción
JComboBox	Lista desplegable

Todas las clases de componentes empiezan por la letra J. Para incorporar alguno de estos componentes en nuestro programa simplemente se debe instanciar un objeto de uno de estas clases. Su uso no difiere del resto de objetos ya vistos en Java. Más adelantes veremos un ejemplo completo de creación de una aplicación que usará interfaz gráfica y se mostrará el código para crear diversos componentes y cómo editarlos a través de sus propiedades y métodos.

Hay un grupo de componentes especiales que sirven para agrupar otros componentes, y poder así organizarlos dentro de nuestra ventana: los contenedores.

5.2 Contenedores

Un contenedor es un tipo especial de componente cuyo propósito es agrupar un conjunto de componentes. Todas las GUI de Swing deben tener como mínimo un contenedor que almacenará el resto de los componentes de nuestra interfaz gráfica. Sin un contenedor, el resto de los componentes no se pueden mostrar.

Swing ofrece varios tipos de contenedores. Podemos clasificarlos en dos tipos: los de nivel superior (o pesados) y los de nivel inferior (o ligeros).

Los contenedores de nivel superior ocupan el primer lugar en la jerarquía, es decir, no se incluyen en ningún otro contenedor. Es más, en nuestra GUI siempre debemos empezar por un

contenedor de nivel superior que incluirá el resto de los componentes.

Entre los contenedores de nivel superior podemos destacar:

- JFrame: implementa una ventana. Sería la ventana principal de nuestra aplicación.
- JDialog: implementa una ventana de tipo diálogo. Serían las ventanas secundarias de nuestra aplicación, que se llamarían desde la ventana principal (JFrame).
- JApplet: implementa una zona dentro de un Applet donde poder incluir componentes de Swing.

El que más usaremos en este curso será JFrame. Podemos destacar varios métodos, de los muchos que tiene la clase JFrame, por el uso que les daremos:

setDefaultCloseOperation(int)	Especifica la acción a realizar cuando cerramos la ventana (pulsando sobre el icono 'X'). Las opciones son: • DO_NOTHING_ON_CLOSE: no hace nada. • HIDE_ON_CLOSE: esconde el JFrame. • DISPOSE_ON_CLOSE: borra el JFrame de memoria. La aplicación puede seguir en funcionamiento si tiene más operaciones ejcutándose. • EXIT_ON_CLOSE: equivale a un System.exit. Es decir, cierra la aplicación entera.
setVisible(boolean)	Hace que la ventana se muestre. Por defecto no lo hace (el booleano es false). Es un método heredado de la clase Window.
setSize(int, int)	Establece el tamaño de la ventana. El primer número especifica el ancho y el segundo el alto. Se puede usar el método pack para ajustar el tamaño al contenido de la ventana. Es un método heredado de la clase Window.
setLocationRelativeTo(Component)	Establece la posición de la ventana relativa al componente especificado. Si se pone "null", la ventana se centra en la pantalla. Es un método heredado de la clase Window.

Los contenedores de nivel inferior, o contenedores ligeros, se usan para organizar grupos de componentes relacionados entre sí. Algunos ejemplos de contenedores ligeros son JPanel, JScrollPane o JRootPane.

A modo de resumen, para tener una idea más clara de cómo funcionan los diferentes tipos de contenedores y cómo se relacionan entre sí, podemos tener una aplicación con una ventana principal, JFrame. Dentro de este contenedor, podríamos tener un JRootPane, que sirve para gestionar otros paneles y

además se encarga de gestionar la barra de menús opcional de nuestra aplicación. Luego podríamos tener otros contenedores ligeros para agrupar diferentes componentes de nuestra aplicación, que se podrían ir mostrando u ocultando según ciertos eventos, como podrían ser seleccionar una opción concreta de la barra de menús.

Para terminar con el concepto de contenedor, debemos conocer los administradores de diseño, que sirven para ubicar los componentes gráficos de nuestra aplicación dentro del contenedor.

En la siguiente tabla vemos los administradores de diseño que nos ofrece Swing.

FlowLayout	Los componentes se ubican de izquierda a derecha y de arriba abajo.
BorderLayout	Los componentes se ubican en el centro o los bordes del contenedor. Divide el contenedor en cinco partes (norte, sur, este, oeste y centro).
GridLayout	Los componentes se organizan en una cuadrícula (filas y columnas).
GridBagLayout	Los componentes se organizan en una cuadrícula flexible (un componente puede ocupar más de una fila o columna).
BoxLayout	Los componentes se organizan horizontal o verticalmente en un cuadro.
SpringLayout	Los componentes se organizan con respecto a una serie de restricciones.

No siempre tendremos que usar uno de los layouts anteriores en nuestra aplicación. Una práctica bastante habitual es la de no usar ningún layout, es decir, usar el layout NULL. Con esto

tenemos completa libertad para ubicar los componentes en la posición que queramos y con el tamaño deseado.

5.3 Controladores de eventos

En todas las aplicaciones hay componentes que reaccionan a algún evento. Por ejemplo, cuando pulsamos un botón, cuando cerramos una ventana, cuando presionamos la tecla Intro sobre un campo de texto, etc.

Java ofrece una serie de controladores de eventos, llamados listeners, especializados en un evento concreto sobre un componente. En estos listeners podemos implementar las acciones que se llevarán a cabo después de dicho evento.

En la siguiente tabla se pueden ver algunos listeners y la acción a la que responden.

ActionListener	Hace referencia a la acción más típica sobre un componente. Por ejemplo, sobre un JButton será presionarlo, sobre un JTextField será pulsar Intro, sobre un JComboBox será seleccionar una opción, etc.
FocusListener	Ejecuta acciones cuando un componente obtiene o pierde el foco (colocarnos sobre el componente o irnos cuando éste estaba activo).
KeyListener	Responde a la acción de pulsar una tecla cuando un componente tiene el foco.
ItemListener	Hace referencia a la acción de seleccionar o deseleccionar una opción (en un JCheckBox, por ejemplo).
MouseListener	Responde al click sobre el ratón.
MouseMotionListener	Responde a acciones como arrastrar (drag) un elemento o pasar por encima.
WindowListener	Responde a acciones sobre una ventana como, por ejemplo, cerrarla.

Para incorporar listeners a nuestro programa, debemos crear primero el componente. Podemos añadir varios listeners a un mismo componente. Por ejemplo, podemos querer que un botón realice una acción al pulsarlo con el ratón, pero también queremos que reaccione al pulsar Intro cuando el botón tenga el foco.