Tema 4:

La librería estándar de Java

Objetivos del tema: *Hasta ahora hemos cubierto lo básico de la sintaxis del lenguaje de programación Java. Pero los lenguajes de programación modernos (a diferencia de C) proporcionan unas librerías estándar que permiten resolver gran parte de las tareas más habituales de programación a las cuales los programadores se van a tener que enfrentar. C también posee una librería estándar, pero tiene un tamaño significativamente inferior al de la librería estándar de Java, o al de la librería estándar de cualquier otro lenguaje moderno. En este tema realizamos una introducción a dos partes importantes de la librería estándar de Java: la librería de entrada y salida, y la librería de programación gráfica.*

INDICE:

1 La librería de entrada y salida

En este primer apartado vamos a abordar el paquete java.io, que contiene la funcionalidad relacionada con las operaciones de entrada y salida (acceso al sistema de ficheros). Acceder a la entrada y salida es fundamental para la construcción de aplicaciones: toda la información que esté almacenada en la memoria RAM del equipo se pierde cuando éste se reinicia. El almacenamiento persistente lo proporciona el sistema de ficheros. Si nuestra aplicación no es capaz de acceder al sistema de ficheros y almacenar información en él, cada vez que se ejecute será igual que si se estuviese ejecutando por primera vez: no será posible recordar nada de ejecuciones anteriores ni permitirá guardar ninguna información al usuario. Obviamente, la mayor parte de las aplicaciones informáticas no funcionan de este modo, sino que requieren guardar algún tipo de información de modo persistente para poder acceder a ella en la siguiente ejecución.

1.1 La clase File

Lo más importante para comprender cómo funciona la clase File es comprender que no es un archivo. Su nombre, francamente, ha sido poco afortunado. File es una forma de referenciar de una ruta en un sistema de ficheros. Esa ruta podría no existir físicamente, es decir, puede que apunte un fichero que no existe. O podría ser la ruta correspondiente con un directorio (una carpeta para los que siempre habéis vivido en sistemas operativos de ventanas), y no con un fichero.

La clase File tiene cuatro constructores; los dos más empleados son:

```
File(String pathname)
File(String parent, String child)
```

Al primero se le pasa la ruta absoluta que queremos que represente el objeto File (por ejemplo, "C:/carpeta/fichero.txt"); al segundo se le pasa el directorio en el cual se sitúa el archivo y el nombre del archivo (por ejemplo, "C:/carpeta" y "fichero.txt"). Habrás podido observar que he empleado como separador de la ruta el carácter "/". En Java puede emplearse como separador tanto el carácter "/" como el carácter "\", y el programa funcionará correctamente en cualquier plataforma, independientemente de qué separador concreto use el sistema operativo. Java se encargará de traducir el separador empleado por el programa al adecuado en el sistema operativo.

De todos modos, debemos recordar una cosa: el carácter "\" es un carácter de escape dentro de una cadena de caracteres en Java (al igual que sucedía en C). Para representar dicho carácter debemos de emplear la secuencia de escape "\\". Por tanto, para representar con este separador la ruta del ejemplo anterior deberemos escribir "C:\\carpeta\\fichero.txt".

Si creamos un objeto File empleando el constructor al cual sólo se le pasa una cadena de caracteres, y esa cadena de caracteres no es una ruta absoluta dentro de nuestro sistema de ficheros, el constructor la tratará como una ruta de acceso relativa. Esta ruta se considera siempre que es relativa al directorio donde se está ejecutando la máquina virtual de la aplicación. Es posible averiguar cuál es el directorio actual de trabajo de la aplicación mediante la sentencia:

```
System.getProperty("user.dir"));
```

A continuación, enumeraremos algunos de los métodos más útiles de la clase **File**:

- **createNewFile**(): si el fichero representado por el objeto **File** no existe, lo crea.
- **delete**(): borra el fichero o directorio representado por este objeto.
- **isDirectory**() **y isFile**(): devuelven true cuando el objeto referencia a un directorio o a un fichero, respectivamente. En caso contrario, devuelven false.
- **mkdir**() y **mkdirs**(): crean el directorio representado por el objeto **File** . El primero, sólo crea un directorio; el segundo creará todos los directorios padres que sean necesarios para crear un fichero cuya ruta coincida con la representada en este objeto.
- **length**(): devuelve el tamaño del fichero referenciado por este objeto.
- **listFiles**(): devuelve un array de objetos **File** conteniendo todos los archivos que estén dentro del directorio representado por el objeto **File** sobre el cual se invocó.
- **canExecute**(), **canRead**() y **canWrite**(): devuelven true cuando tenemos permiso para realizar la operación de ejecución, lectura o escritura sobre el fichero correspondiente y false en caso contrario.

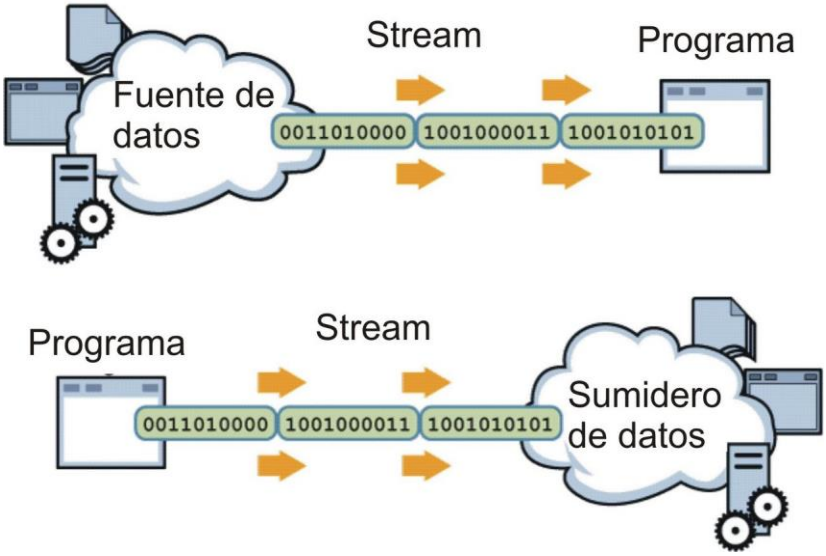
La clase **File** posee muchos otros métodos. Te recomiendo que le eches un vistazo a su javadoc para familiarizarte con ellos.

En el siguiente ejemplo vemos un programa que contiene una cadena de caracteres que representa un directorio en nuestro sistema de ficheros. El programa lista todos los archivos contenidos en dicho directorio y, para cada uno de ellos, muestra una serie de información como su nombre, su ruta absoluta, sus permisos de lectura, escritura y ejecución, etcétera.

```
import java.io.*;
import java.util.*;

public class Ejemplo1 {
  public static void main(String args[]) throws IOException {
    File directorio = new File("C:/");
    if ( (directorio.exists()) && (directorio.isDirectory()))
{
      File[] lista = directorio.listFiles();
      for (int i = 0; i < lista.length; i++) {
        System.out.println(lista[i].getAbsolutePath());
        System.out.println("Nombre: " + lista[i].getName());
        System.out.println("Ruta absoluta " +
lista[i].getAbsolutePath());
        System.out.println("Ruta: " + lista[i].getPath());
        System.out.println("Padre: " + lista[i].getParent());
        System.out.println("¿Puedo leerlo? " +
lista[i].canRead());
        System.out.println("¿Puedo escribirlo? " +
lista[i].canWrite());
        System.out.println("Tamaño en bytes: " +
lista[i].length());
        System.out.println("Fecha de la última modificación:
" +
                           new
Date(lista[i].lastModified()));
        System.out.println("\n");
      }
    }
    else {
      System.out.println("El directorio no existe");
    }
  }
}
```

1.2 Flujos de datos

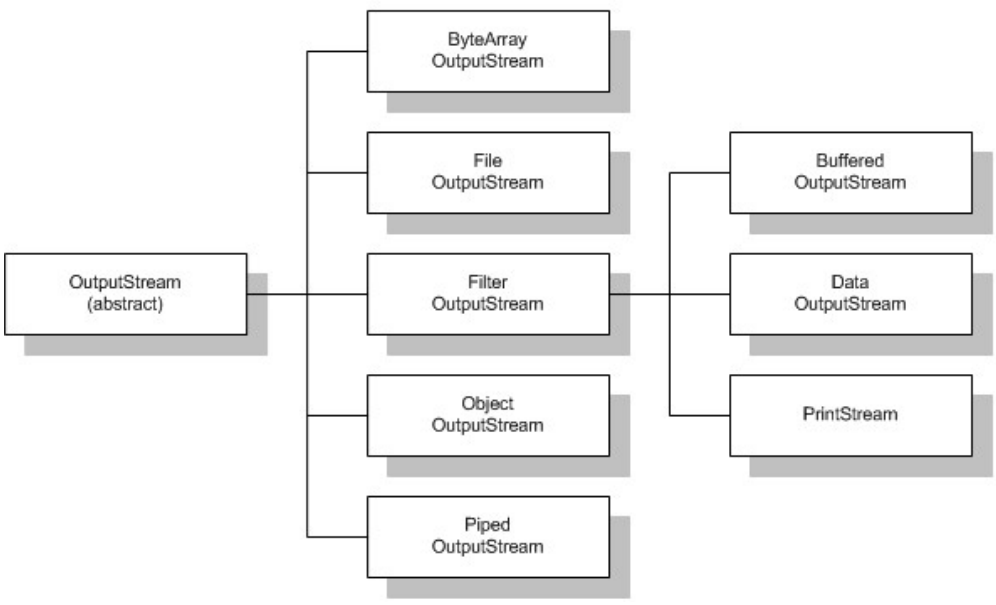
Los flujos de datos (***streams*** en inglés) son una abstracción empleada en muchos lenguajes de programación, entre ellos Java, para representar cualquier fuente que produzca o consuma información. Su nombre (flujo) viene de que pueden considerarse como una cadena de datos continúa con una longitud que, posiblemente, es desconocida, al igual que la "longitud" de un flujo de agua que está corriendo. Los flujos de datos o ***streams*** representan cualquier fuente que proporcione datos al programa, o cualquier sumidero que tome datos del programa



Existen dos tipos de flujos de datos: los binarios o de bytes, y los de texto. En los primeros, como su nombre indica, la información que fluye está en formato binario, mientras que en los segundos la información es texto. Cada una de estas dos categorías puede volver a subdividirse en flujos de datos de entrada y flujos de datos de salida. Los primeros serían flujos que nos proporcionan datos, es decir, entradas de nuestro programa. Los segundos serían flujos en los cuales nuestro programa escribe datos, es decir, salidas de nuestro programa.

En este apartado vamos a ver las principales clases para representar flujos binarios y de texto, de entrada y de salida que proporciona Java.

Flujos de salida de bytes



En la figura podemos ver la jerarquía de los flujos de salida de bytes de Java. Organizar de

modo jerárquico las clases que permiten acceder a la funcionalidad de entrada y salida va a ser un patrón que veremos varias veces a lo largo de este apartado. Como podemos observar en la figura, la clase padre de todos los flujos de salida de Java es **OutputStream**. Se trata de una clase abstracta (por tanto, no vamos a poder crear objetos de ella porque su funcionalidad está "incompleta") que representa un flujo de datos de salida binario cualquiera. Sus métodos son los siguientes:

- close(): cierra el flujo de datos y libera cualquier recurso que el flujo de datos pudiera estar consumiendo. Por ejemplo, este método permite liberar los recursos del sistema operativo consumidos por un fichero al cual hemos terminado de escribir información.
- flush(): sincroniza este flujo de datos con el dispositivo al cual se están escribiendo los datos. Este método es necesario porque, habitualmente, los **OutputStream** tienen un buffer de datos interno al cual van escribiendo la información antes de enviarla al dispositivo de entrada y salida correspondiente. De este modo, se pueden agrupar varias operaciones de escritura y efectuarlas de un modo más eficiente. Este método hace que los últimos cambios realizados sobre el buffer de memoria se sincronicen con el dispositivo de entrada y salida.
- write(byte[] b): escribe el array b de bytes que se le pasa como argumento al flujo de salida.
- write(byte[] b, int off, int len): escribe **len** bytes del array **b** al flujo de salida, empezando a escribirlos en el offset indicado por **off** .
- abstract void write(int b): escribe 1 byte al flujo de salida.

Dado que la clase **OutputStream** es abstracta nunca vamos a poder crear un objeto de ella. Tendremos que crear objetos de alguna de sus subclases que, como podemos ver en la figura 2, son bastantes. Resumamos, brevemente, cuál es el propósito de cada una de ellas:

- **ByteArrayOutputStream**: como su nombre indica, este flujo de salida representa un array de bytes que se almacena en memoria. A partir de él puede obtenerse una cadena de caracteres que represente todos los datos escritos. Obviamente, no soluciona el problema de almacenar información de modo persistente.
- **FileOutputStream**: flujo de salida para la escritura de datos a un objeto de tipo **File**.

- **FilterOutputStream**: esta clase encapsula a otro objeto de tipo **OutputStream** e intercepta todas las operaciones de escritura para, posiblemente, realiza alguna transformación sobre los datos que se están escribiendo. Una de sus subclases, **DataOutputStream** resulta particularmente útil para escribir distintos tipos de datos primitivos a un flujo de datos de salida.
- **ObjectOutputStream**: encapsula otro objeto de tipo **OutputStream** y permite escribir objetos Java completos al flujo de datos de salida representado por el **OutputStream** correspondiente.
- **PipedOutputStream**: junto con la clase **PipedInputStream** permite emplear pipes para comunicar (habitualmente) dos **threads**.

Como podemos observar, cada una de las clases hijas de **OutputStream** proporciona una funcionalidad más específica a la clase padre. Dos de ellas (**FilterOutputStream** y **ObjectOutputStream**) actúan como decoradores sobre cualquier **OutputStream**; es decir, encapsulan un objeto de tipo **OutputStream** y le proporcionan funcionalidad adicional (aplicar algún tipo de filtrado sobre los datos, o permitir escribir objetos Java).

Cuando queremos escribir datos en binario a un archivo lo más habitual es comenzar creando un objeto **File** que represente a dicho archivo y creando un flujo de datos de salida mediante la clase FileOutputStream:

```
File f2 = new File ("C:/datos.dat");
FileOutputStream out = new FileOutputStream(f2);
```

El siguiente paso depende de qué tipo de datos queramos escribir. Hay dos escenarios habituales: queremos escribir tipos de datos primitivos (float, double, boolean...), o queremos escribir objetos Java completos. En el primer caso, debemos emplear la clase **DataOutputStream**; al crear el objeto de esta clase le pasaremos a su constructor el **FileOutputStream** que representa el fichero al cual queremos escribir los tipos de datos primitivos:

```
DataOutputStream out = new DataOutputStream(fout);
```

DataOutputStream tiene un conjunto de métodos con nombre **writeXXX** donde XXX son los nombres de los distintos tipos de datos primitivos existentes en Java. Cada uno de esos métodos permite escribir el tipo de dato primitivo correspondiente al flujo de datos de salida:

```
out.writeBoolean(true);
out.writeInt(45);
out.writeDouble(4.8);
```

Si lo que queremos escribir son objetos Java, debemos emplear la clase **ObjectOutputStream** y, al crear el objeto de esta clase, le pasaremos a su constructor el **FileOutputStream** que representa el fichero al cual queremos escribir los objetos:

```
FileOutputStream fout2 = new FileOutputStream(f2);
ObjectOutputStream out2 = new ObjectOutputStream(fout2);
out2.writeObject(new Date());
```

la última sentencia está escribiendo un objeto de tipo **java.util.Date** al fichero representado por el **File** f2 .

Finalmente, siempre que terminemos de trabajar con un flujo de datos debemos cerrarlo para liberar los recursos que dicho flujo de datos está consumiendo. Esto se hace invocando a su método close() .

En el siguiente ejemplo podemos ver un código que hace uso tanto de un **DataOutputStream** como de un **ObjectOutputStream**. Ambos están asociados con un **File** que se creará en C:, el nombre del primer fichero será datos.dat y el del segundo objetos.dat. Observa como las operaciones de creación de los flujos de datos y las escrituras están dentro de un **bloque try-catch**. Si se produce una excepción, mostraremos un mensaje de error por la consola. En la cláusula finally cerramos los ficheros y los flujos de datos. Observa que estas sentencias también están dentro de un **bloque try-catch**, ya que también lanzan excepciones, que en el caso que nos atañe, son excepciones del tipo **IOException** (**Input-Output Exception**). Presta especial atención al modo de cerrar todos los recursos que, dependiendo de los distintos cauces de ejecución del programa (es decir, de si se ha lanzado alguna excepción o no), pueden haberse adquirido durante su ejecución.

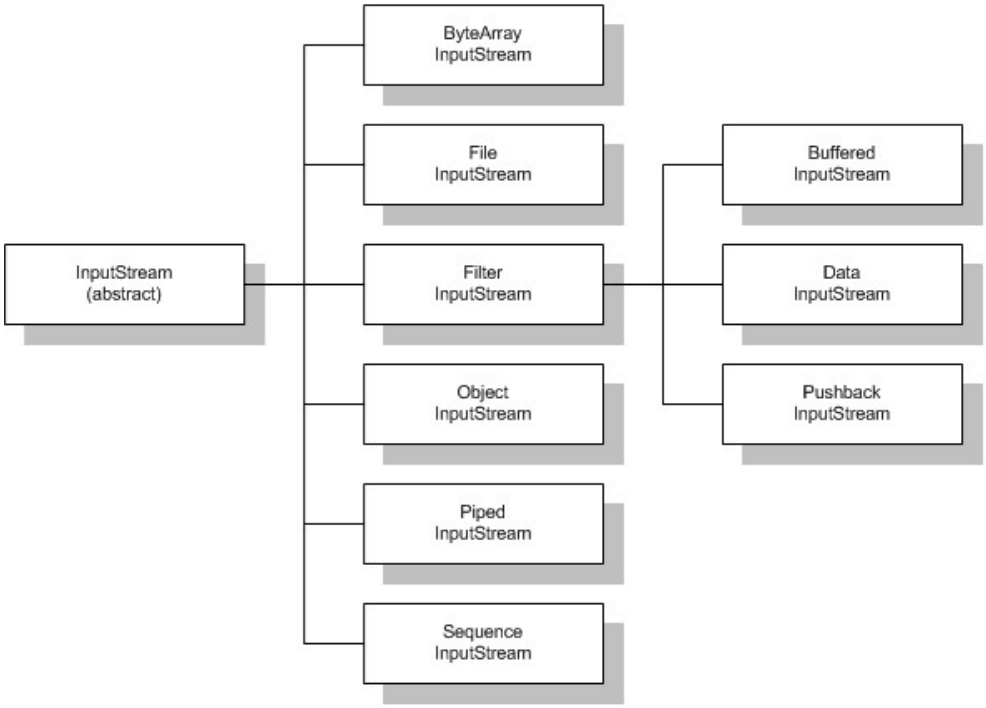
```
import java.io.*;
import java.util.*;

public class Ejemplo2 {

    static public void main(String[] args) {
        File file1 = new File("C:/datos.dat");
```

```
        File file2 = new File("C:/objetos.dat");
        FileOutputStream fileOutputStream = null,
fileOutputStream2 = null;
        DataOutputStream dataOutputStream = null;
        ObjectOutputStream objectOutputStream = null;

        try {
            fileOutputStream = new FileOutputStream(file1);
            dataOutputStream = new
DataOutputStream(fileOutputStream);

            dataOutputStream.writeBoolean(true);
            dataOutputStream.writeInt(45);
            dataOutputStream.writeDouble(4.8);
            fileOutputStream2 = new FileOutputStream(file2);
            objectOutputStream = new
ObjectOutputStream(fileOutputStream2);
            objectOutputStream.writeObject(new Date());

        } catch (IOException ex) {
            System.out.println("Se ha producido un error
antes de terminar las escrituras");
            ex.printStackTrace();
        } finally {
            if (dataOutputStream != null) {
                try {
                    dataOutputStream.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
            if (fileOutputStream != null) {
                try {
                    fileOutputStream.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
            if (objectOutputStream != null) {
                try {
                    objectOutputStream.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
            if (fileOutputStream2 != null) {
                try {
                    fileOutputStream2.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
```

```
        }
}
```

Flujos de entrada de bytes

Ya sabemos cómo enviar información en binario a un archivo. Ahora necesitamos aprender cómo recuperarla. En la siguiente figura podemos ver la jerarquía de los flujos de entrada de bytes de Java. Como podemos observar en la figura, la clase padre de todos los flujos de entrada de Java es InputStream. Se trata de una clase abstracta que representa un flujo binario de datos de salida.



Los métodos de InputStream son los siguientes:

- close(): cierra el flujo de datos y libera cualquier recurso que el flujo pudiera estar consumiendo.
- int available(): devuelve una estimación del número de bytes que se pueden leer de este flujo de datos de entrada sin producirse un bloqueo.
- int read(byte[] b) : lee los bytes que haya disponibles en este flujo de entrada, leyendo nunca más del tamaño del array **b**, y los almacena en este array. Devuelve el número de bytes que ha leído correctamente. Devuelve -1 si se ha alcanzado el final del flujo de entrada. Si no hubiera ningún byte disponible el método espera hasta que haya datos que se puedan leer; esto hace que se detenga el programa que invoca este método hasta que haya

datos (realmente sólo se bloquea el thread que ejecuta el método, no todo el programa).

- int read(): funciona exactamente del mismo modo que el anterior método, a excepción de que lee un único byte.
- abstract int read(byte[] b, int off, int len): lee hasta **len** bytes del flujo de entrada y los almacena en el array b , empezando a leer los datos en el offset indicado por off. Su comportamiento es similar al de los métodos anteriores.
- skip(long n): ignora los próximos **n bytes** del flujo de entrada.

Dado que la clase **InputStream** es abstracta siempre tendremos que usar alguna de sus clases hijas. Las clases hijas más comunes son ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream y PipedInputStream. ¿Te comienzan a resultar familiares los nombres?. Efectivamente, todas estas clases son las recíprocas de los flujos de salida. Nuevamente, las más empleadas son **ObjectInputStream y DataInputStream**. La primera se emplea para leer objetos de un flujo de entrada. La segunda es una subclase de **FilterInputStream** y se emplea para leer todo tipo de datos primitivos; para ello se emplean un conjunto de métodos **readXXX**, donde XXX son los nombres de todos los tipos de datos primitivos existentes en Java. Estos métodos leen el correspondiente dato primitivo del flujo de entrada y lo proporcionan como dato de retorno.

La forma más habitual de crear objetos de estas dos clases es partiendo de un objeto **FileInputStream**; ambas clases "envuelven" al objeto FileInputStream y proporcionan funcionalidad adicional (lectura de tipos de datos primitivos, o de objetos completos) a él. El **FileInputStream** se puede crear directamente pasándole un objeto de tipo File en su constructor.

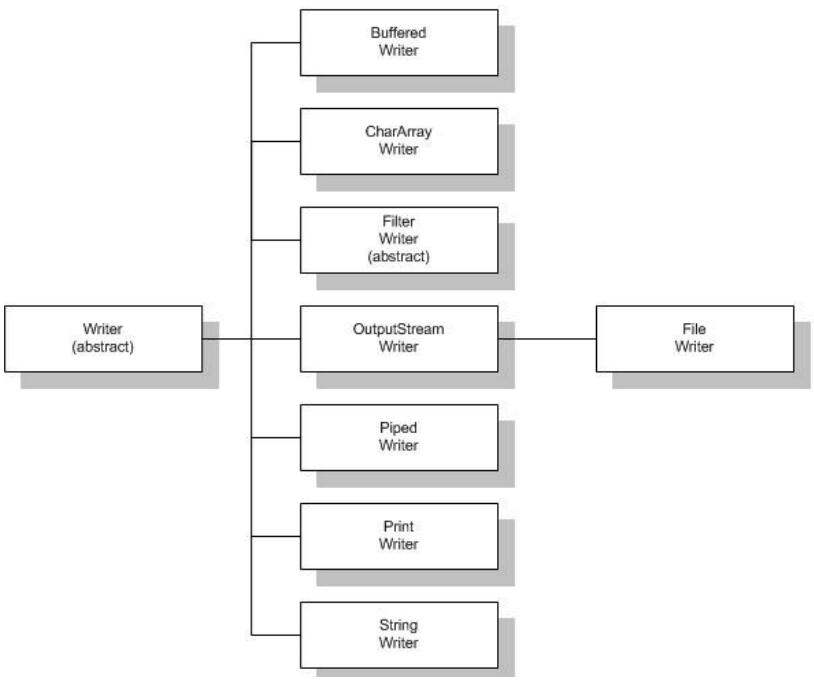
En el Ejemplo3 vemos un programa que permite abrir los archivos generados por el código del Ejemplo2 y muestra su contenido por consola. La fecha que se mostrará no es la fecha actual, si no la fecha que se escribió en el fichero cuando se creó. La capacidad de Java para escribir un objeto completo con una sola instrucción, sin importar lo complejo que sea el objeto, es muy atractiva. Nos permite almacenar una gran cantidad de información de un modo muy sencillo. La desventaja de este mecanismo de persistencia es que el formato en el que se guarda la información es binario y es muy complejo acceder a esa información desde otro lenguaje de programación. Es más, si modificamos el código fuente de la clase que hemos serializado probablemente cuando intentemos leer un objeto de esa clase no lo vamos a recuperar de modo correcto. Por ello, por lo general, no es recomendable emplear esta estrategia para almacenar información a largo plazo; aunque sí

resulta muy práctica para almacenar información de modo temporal (por ejemplo, para mantener copias de respaldo de la sesión de trabajo del usuario).

```
import java.io.*;

public class Ejemplo3 {

    static public void main(String[] args) {
        File file1 = new File("C:/a/datos.dat");
        File file2 = new File("C:/a/objetos.dat");
        FileInputStream fileInputStream = null,
fileInputStream2 = null;
        DataInputStream dataInputStream = null;
        ObjectInputStream objectInputStream = null;

        try {
            fileInputStream = new FileInputStream(file1);
            dataInputStream = new
DataInputStream(fileInputStream);


System.out.println(dataInputStream.readBoolean());
            System.out.println(dataInputStream.readInt());
            System.out.println(dataInputStream.readDouble());
            fileInputStream2 = new FileInputStream(file2);
            objectInputStream = new
ObjectInputStream(fileInputStream2);

System.out.println(objectInputStream.readObject());

        } catch (Exception ex) {
            System.out.println("Se ha producido un error
antes de terminar las escrituras");
            ex.printStackTrace();
        } finally {
            …
            }
        }
    }
}
```

1.3 Flujos de salida de texto

Como norma general, almacenar información de modo binario es más simple que almacenarla en modo



texto. Además, la información en binario suele ser mucho más compacta (ocupa menos espacio) que en modo texto. Sin embargo, la información que se almacena en modo texto es mucho más fácil de leer desde otros lenguajes de programación; no tendremos problemas si una clase de la cual estamos serializando objetos cambia y el archivo que generemos será comprensible para un ser humano.

En la figura mostramos la jerarquía de clases de los flujos de salida de texto en Java. En lo alto de la jerarquía tenemos, nuevamente, una clase abstracta: **Writer** . Esta clase representa cualquier flujo de datos de salida donde la información se va a representar en modo texto. Sus métodos principales son:

- close(): cierra el flujo de datos y libera cualquier recurso que el flujo pudiera estar consumiendo.
- flush(): sincroniza este flujo de datos con el dispositivo al cual se están escribiendo los datos.
- write(byte[] b): escribe el array **b** de bytes que se le pasa como argumento al flujo de salida.
- write(byte[] b, int off, int len): escribe **len** bytes del array **b** al flujo de salida, empezando a escribirlos en el offset indicado por **off** .
- abstract void write(int b): escribe 1 byte al flujo de salida.

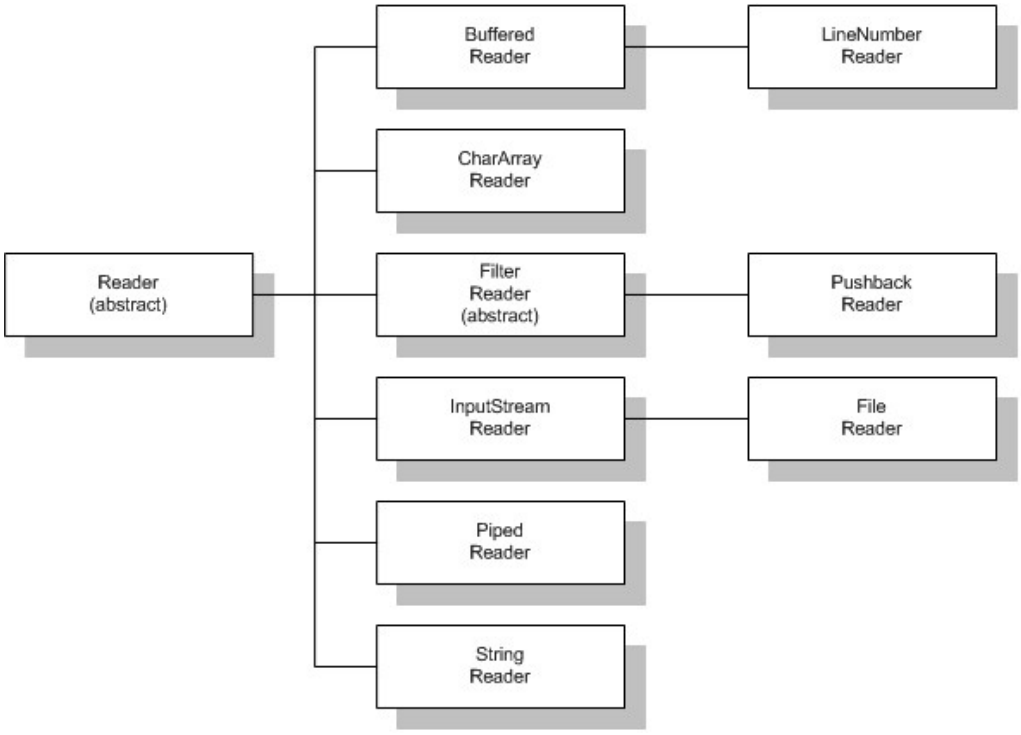
Supongo que habrás visto el paralelismo con los métodos de **OutputStream**. Nuevamente, la clase Writer no se puede emplear directamente, sino que tenemos que usar alguna de sus subclases:

- BufferedWriter: esta clase es básicamente idéntica a Writer, sólo que no realiza las operaciones de escritura directamente sino que las almacena en un buffer para aumentar su eficiencia.
- CharArrayWriter: representa un flujo de salida constituido por un array en memoria.
- FilterWriter: envuelve a un objeto de tipo **Writer** y aplica un conjunto de filtros a las operaciones de escritura que se realizan sobre él.
- OutputStreamWriter: esta clase es una especie de puente entre flujos binarios y flujos de caracteres. Todos los caracteres que se escriban a ella serán almacenados como bytes empleando un cierto charset para codificarlos.
- PipedWriter: junto con la clase **PipeReader**, suele emplearse para comunicar dos thread que quieren compartir información.

- StringWriter: todas las escrituras a los objetos de esta clase se almacenan en memoria, y a partir del objeto es posible recuperar una cadena de caracteres con todo el texto escrito.
- PrintWriter: esta clase es bastante similar a **DataOutputStream** ; contiene un conjunto de métodos **print** y **println** que permiten escribir todos los tipos de datos primitivos de Java en modo texto al flujo de datos de salida. Para ello, obviamente, ambos métodos están sobrecargados (existen muchas versiones de los métodos que se diferencian en el tipo de parámetro que se le pasa). PrintWriter: posiblemente sea la clase más empleada para construir archivos de texto. Podemos construir una instancia de ella proporcionando un objeto **File** en su constructor, o también un objeto de tipo OutputStream.

Flujos de entrada de texto

En la figura podemos ver la jerarquía de clases que permiten manipular flujos de entrada de texto en Java.



La clase abstracta que está en la cima, Reader, representa un flujo de entrada de texto cualquiera y sus métodos más importantes son:

- close(): cierra el flujo de datos y libera cualquier recurso que el flujo pudiera estar consumiendo.

- int read(char[] b): lee los datos que haya disponibles en este flujo de entrada, leyendo nunca más del tamaño del array b, y los almacena en este array. Devuelve el número de caracteres que ha leído correctamente. Devuelve -1 si se ha alcanzado el final del flujo de entrada. Si no hubiera ningún dato disponible, el método espera hasta que haya datos que se puedan leer.
- int read(): funciona exactamente del mismo modo que el anterior método, a excepción de que lee un único **char**.
- abstract int read(char [] b, int off, int len): lee hasta **len** caracteres del flujo de entrada y los almacena en el array **b**, empezando a leer los datos en el offset indicado por **off** . Su comportamiento es similar al de los métodos anteriores.
- boolean ready(): devuelve **true** si hay caracteres disponibles para leer, y **false** en caso contrario.
- skip(long n): ignora los próximos **n** bytes del flujo de entrada.

Las clases concretas, hijas de la clase **Reader**, que se emplean para la lectura de datos son BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader y StringReader. Son las clases recíprocas de los hijos de Writer. Una forma bastante común de leer un archivo de texto en Java es crear un objeto de tipo FileInputStream a partir de un objeto de tipo **File** que lo represente; con este objeto creamos un InputStreamReader y éste, a su vez, se lo pasamos a un **BufferedReader**.

BufferedReader tiene un método, readLine(), que devuelve una línea de texto del flujo de datos de entrada. Si no hay ninguna línea de texto disponible en ese momento, se bloquea y espera a que haya texto disponible. Si hemos llegado al final del flujo de datos, devuelve **null**.

En el Ejemplo4 vemos un código Java que abre un fichero de texto con nombre "original.txt" (el fichero debe estar situado en la raíz de nuestro disco C:) y, en el mismo directorio, crea una copia del fichero con nombre "copia.txt".

```
import java.io.*;
public class Ejemplo4 {

    public static void main(String[] args) {

        File file1 = new File("C:/original.txt");
        File file2 = new File("C:/copia.txt");
        FileInputStream fileInputStream = null;
        InputStreamReader inputStreamReader = null;
```

```
        BufferedReader bufferedReader = null;
        PrintWriter printWriter = null;
        try {
            fileInputStream = new FileInputStream(file1);
            inputStreamReader = new
InputStreamReader(fileInputStream);
            bufferedReader = new
BufferedReader(inputStreamReader);
            String texto = bufferedReader.readLine();
            printWriter = new PrintWriter(file2);

            do {
                System.out.println(texto);
                printWriter.println(texto);
            } while ((texto = bufferedReader.readLine()) !=
null);
        } catch (IOException ex) {
            System.out.println("Error durante el proceso de
copia");
            ex.printStackTrace();
        } finally {
            ...
    }
}
```

La técnica empleada para leer y escribir documentos de texto del listado 4 funciona perfectamente mientras toda la información que manipulamos sean cadenas de caracteres. Pero ¿qué pasa si tenemos que trabajar con datos numéricos, por ejemplo, que están en formato de texto? Estos datos vamos a tener que leerlos inicialmente como cadenas de caracteres, y después deberemos transformar la cadena de caracteres al tipo de dato adecuado.

Para realizar esta transformación podemos apoyarnos en un conjunto de clases "wraper" de los tipos de datos primitivos que podemos encontrar en el paquete **java.lang**. Estas clases son **Boolean, Byte, Short, Character, Integer, Long, Float y Double**. Cada una de ellas encapsula una instancia del tipo de dato primitivo correspondiente. Cada una de estas clases tiene un método con nombre de **parseXXX** (String s), donde XXX es el nombre de la propia clase. A este método se le pasa una cadena de caracteres que contiene un valor del tipo primitivo correspondiente y nos devuelve el valor representado en la cadena como un tipo de dato primitivo. Por ejemplo, la sentencia:

```
Double d = Double.parseDouble("3.4");
```

almacenará el valor 3.4 en la variable d. Si el formato de la cadena de caracteres que le pasamos no se corresponde con el tipo de dato que espera el método (por ejemplo, si le pasamos al método anterior la cadena de caracteres "hola") el método lanzará una excepción. En el Ejemplo5 podemos ver un código que escribe varios tipos de datos primitivos, como texto, a un fichero y los vuelve a leer.

```
import java.io.*;

public class Ejemplo5 {

    public static void main(String[] args) {

        File file = new File("C:/datos.txt");
        FileInputStream fileInputStream = null;
        InputStreamReader inputStreamReader = null;
        BufferedReader bufferedReader = null;
        PrintWriter printWriter = null;
        try {

            printWriter = new PrintWriter(file);
            printWriter.println(25);
            printWriter.println(2.5);
            printWriter.println(true);
            printWriter.println(3.6F);
            printWriter.close();
            fileInputStream = new FileInputStream(file);
            inputStreamReader = new
InputStreamReader(fileInputStream);
            bufferedReader = new
BufferedReader(inputStreamReader);
            int entero =
Integer.parseInt(bufferedReader.readLine());
            double realDoble =
Double.parseDouble(bufferedReader.readLine());
            boolean logico =
Boolean.parseBoolean(bufferedReader.readLine());
            float real =
Float.parseFloat(bufferedReader.readLine());
            System.out.println(entero + " " + realDoble + " "
+ logico + " " + real);

        } catch (IOException ex) {
            System.out.println("Error durante el proceso");
            ex.printStackTrace();
        } finally {
            ...
    }
}
```

Si quisiésemos almacenar un objeto completo como texto tendríamos que ir almacenando todos sus atributos como texto. Aquellos atributos que sean tipos de datos primitivos pueden transformarse directamente a una cadena de caracteres. Si algunos de los atributos del objeto vuelven a ser objetos, deberemos guardar todos los atributos del objeto-atributo como texto, y así sucesivamente. Como podéis intuir, este proceso puede llegar a ser bastante tedioso. Nada que ver con lo simple y sencillo que resulta serializar un objeto en formato binario. Eso sí, el fichero de salida será mucho más portable entre lenguajes y diferentes versiones de la plataforma Java.

1.4 Leyendo datos de la consola

System es una clase de la librería estándar de Java, y out es un atributo estático de dicha clase cuyo tipo de dato es **PrintWriter**, una clase hija de FilterOutputStream que proporciona métodos sobrecargados para escribir todos los tipos de datos primitivos de Java. Estos métodos se llaman **println**. También existen un conjunto de métodos equivalentes a éste, pero con nombre print, que no imprimen un retorno de carro después de imprimir el dato. **out** representa el flujo de datos asociado con la salida estándar del sistema.

Para leer datos de la consola, el atributo estático in, de la clase **System**, cuyo tipo es InputStream, se suele envolver en un **InputStreamReader**. in representa el flujo de entrada asociado con la entrada estándar del sistema Después debemos envolverlo en un **BufferedReader**, y a continuación proceder del mismo modo que hemos procedido en el Ejemplo. En el Ejemplo6 podemos ver un código que lee exactamente los mismos datos que el código de Ejemplo5, sólo que en vez de leerlos de un fichero lo hace de la entrada estándar, es decir, de **System.in**.

```
import java.io.*;

public class Ejemplo6 {

    public static void main(String[] args) {
        InputStreamReader inputStreamReader = null;
        BufferedReader bufferedReader = null;
        try {
            inputStreamReader = new
InputStreamReader(System.in);
            bufferedReader = new
BufferedReader(inputStreamReader);
            int entero =
Integer.parseInt(bufferedReader.readLine());
            double realDoble =
Double.parseDouble(bufferedReader.readLine());
```

```
            boolean logico =
Boolean.parseBoolean(bufferedReader.readLine());
            float real =
Float.parseFloat(bufferedReader.readLine());
            System.out.println(entero + " " + realDoble + " "
+ logico + " " + real);

        } catch (IOException ex) {
            System.out.println("Error durante el proceso");
            ex.printStackTrace();
        } finally {
            ...
    }
}
```

Por último, y para demostrar como de unificada está toda la entrada y salida dentro de Java veremos que desde Java es tan simple (o complicado) leer datos a través de la red como leerlos de la consola, mostramos en el Ejemplo7 un pequeño programa que abre un socket y espera a recibir datos por él. Para ello se emplea el método accept() de la clase ServerSocket; este método espera hasta que se realiza algún intento de conexión al puerto donde el **ServerSocket** está escuchando (el 8081 en nuestro ejemplo). Cuando se realiza un intento de conexión el método devuelve un objeto tipo **Socket**, al cual le podemos pedir un **InputStream**, para leer los datos que lleguen al socket (datos que envía el equipo que se ha conectado), y un **OutputStream**, para escribir datos en el socket y enviarlos al equipo que ha establecido la conexión. Tanto la clase **ServerSocket** como la clase **Socket** se encuentran en el paquete java.net. Como podéis ver en el código, la forma de tratar con estos flujos de entrada y de salida es idéntica a la forma de tratar con el flujo de entrada y de salida de un fichero del disco duro, o de la consola.

```
import java.io.*;
import java.net.*;

public class Ejemplo7 {

    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        Socket socket = null;
        DataOutputStream dataOutputStream = null;
        DataInputStream dataInputStream = null;
        try {
            serverSocket = new ServerSocket(8081);
            socket = serverSocket.accept();
            OutputStream os = socket.getOutputStream();
            InputStream is = socket.getInputStream();
            dataInputStream = new DataInputStream(is);
            System.out.println(dataInputStream.readLine());
            System.out.println(dataInputStream.readInt());
            dataOutputStream = new DataOutputStream(os);
```

```
            dataOutputStream.writeBytes("Hola que tal\n");
            dataOutputStream.writeFloat(2.3F);

        } catch (Exception ex) {
            System.out.println("Fallo en la red");
            ex.printStackTrace();
        } finally {
            ...
        }
    }
}
```

En el Ejemplo8 vemos un pequeño programa cliente que intenta conectarse al localhost (la IP 127.0.0.1). Una vez establecida la conexión, se obtienen los objetos **InputStream** y **OutputStream** vinculados con este socket y se emplean para mandar una cadena de caracteres y un número entero. El socket espera recibir una cadena de caracteres y un número real del servidor. Ambos programas pueden ser ejecutados en dos máquinas distintas simplemente cambiando en el código fuente la IP del localhost por la IP de máquina en la que corra el programa servidor (el de Ejemplo7).

```
import java.net.Socket;
import java.io.*;

public class Ejemplo8 {

    public static void main(String[] args) {
        Socket socket = null;
        DataOutputStream dataOutputStream = null;
        DataInputStream dataInputStream = null;
        OutputStream dutputStream = null;
        try {
            socket = new Socket("127.0.0.1", 8081);
            dutputStream = socket.getOutputStream();
            dataOutputStream = new
DataOutputStream(dutputStream);
            dataOutputStream.writeBytes("Hola que tal; yo soy
el cliente\n");
            dataOutputStream.writeInt(23);
            InputStream is = socket.getInputStream();
            dataInputStream = new DataInputStream(is);
            System.out.println(dataInputStream.readLine());
            System.out.println(dataInputStream.readFloat());
        } catch (Exception ex) {
            System.out.println("Fallo en la red");
            ex.printStackTrace();
        } finally {
            ...
        }
    }
}
```

Para que los programas Ejemplo7 y Ejemplo8 funcionen de modo correcto, primero hay que lanzar el programa servidor (el del Ejemplo7) y después el cliente (el del Ejemplo8).

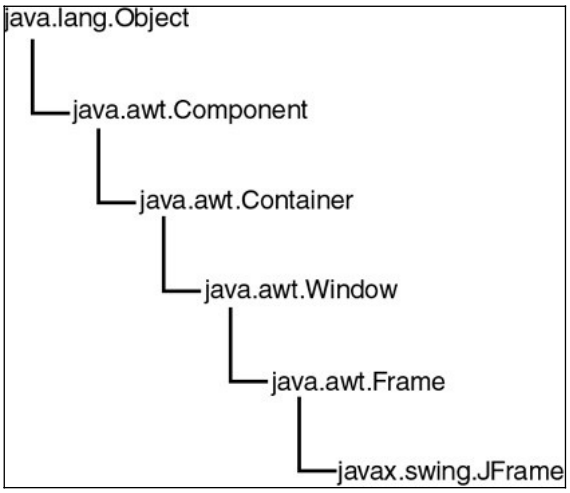
2 Programación gráfica con swing

Hace muchos años que dejó de ser aceptable que el usuario final de una aplicación emplease una consola y el teclado para interaccionar con ella. Ahora, las expectativas de los usuarios han crecido y esperan bonitas interfaces de usuario con botones, menús, barras de tareas, etcétera. El objetivo de este apartado es enseñar a diseñar interfaces gráficas de usuario empleando para ello la librería Swing.

Swing es la librería gráfica que en Java 1.2 sustituyó a la vieja AWT. La nueva librería cuenta con más componentes y proporciona una mayor cantidad de opciones sobre ellos (como distintas apariencias, control sobre el focus, mayor control sobre su aspecto, mayor facilidad para pintar al hacer el buffering transparente al usuario, ...) que su antecesor. Además, se diferencia radicalmente de ésta en su implementación. En AWT cuando añadíamos, por ejemplo, un botón a nuestra interfaz la máquina virtual le pedía al sistema operativo la creación de un botón en un determinado sitio con unas determinadas propiedades; en Swing ya no se pide al sistema operativo nada: se dibuja el botón sobre la ventana en la que lo queríamos. Con esto se eliminaron muchos problemas que existían antes con los códigos de las interfaces gráficas: debido a que dependían del sistema operativo para obtener sus componentes gráficos, era necesario testar los programas en distintos sistemas operativos, pudiendo tener distintos bugs en cada uno de ellos.

Esto, evidentemente, iba en contra de la filosofía de Java, supuestamente un lenguaje que no dependía de la plataforma. Con Swing se mejoró bastante este aspecto: lo único que se pide al sistema operativo es una ventana; una vez que tenemos la ventana dibujamos botones, listas, scroll-bars... y todo lo que necesitemos sobre ella. Evidentemente esta aproximación gana mucho en lo que a independencia de la plataforma se refiere. Además, el hecho de que el botón no sea un botón del sistema operativo sino un botón pintado por Java nos da un mayor control sobre su apariencia.

2.1 JFrame



Graduación de herencia de JFrame 1

Es el contenedor que emplearemos para situar en él todos los demás componentes que necesitemos para el desarrollo de la interface de nuestro programa.

En el gráfico 1 se muestra la jerarquía de herencia de este componente desde **Object**, que como ya dijimos es el padre de todas las clases de Java. Los métodos de este componente estarán repartidos a lo largo de todos sus ascendientes, cosa que hemos de tener en cuenta cuando consultemos la ayuda on-line de esta clase. Así por ejemplo resulta intuitivo que debiera haber un método para cambiar el color de fondo del **frame**, pero él no tiene ningún método para ello, lo tiene **Component**. Veamos el código necesario para crear un JFrame:

```
package ejemplosgui;
import javax.swing.*;

class Frame1 extends JFrame {

    public Frame1() {
        setTitle("Hola!!!");
        setSize(300, 200);
    }
}

public class Ejemplo1 {
    public static void main(String[] args) {
        JFrame frame = new Frame1();
        frame.setVisible(true);
    }
}
```

Sin embargo, nuestro código anterior tiene un problema: al cerrar la ventana no se detiene la máquina virtual. La única forma de acabar con ella será mediante ^c si hemos ejecutado el programa desde una consola o con Control-Alt-Supr y eliminando su tarea correspondiente si lo ejecutamos desde Windows. ¿Por qué sucede esto? porque no hemos escrito el código necesario para ello. Para terminar la máquina virtual hemos de escribir un código que escuche los eventos de ventana, y que ante el evento de intentar cerrar la ventana realice esta acción (terminar la máquina virtual). Antes de seguir con componentes de la librería Swing veamos que es un evento y cómo gestionarlos.

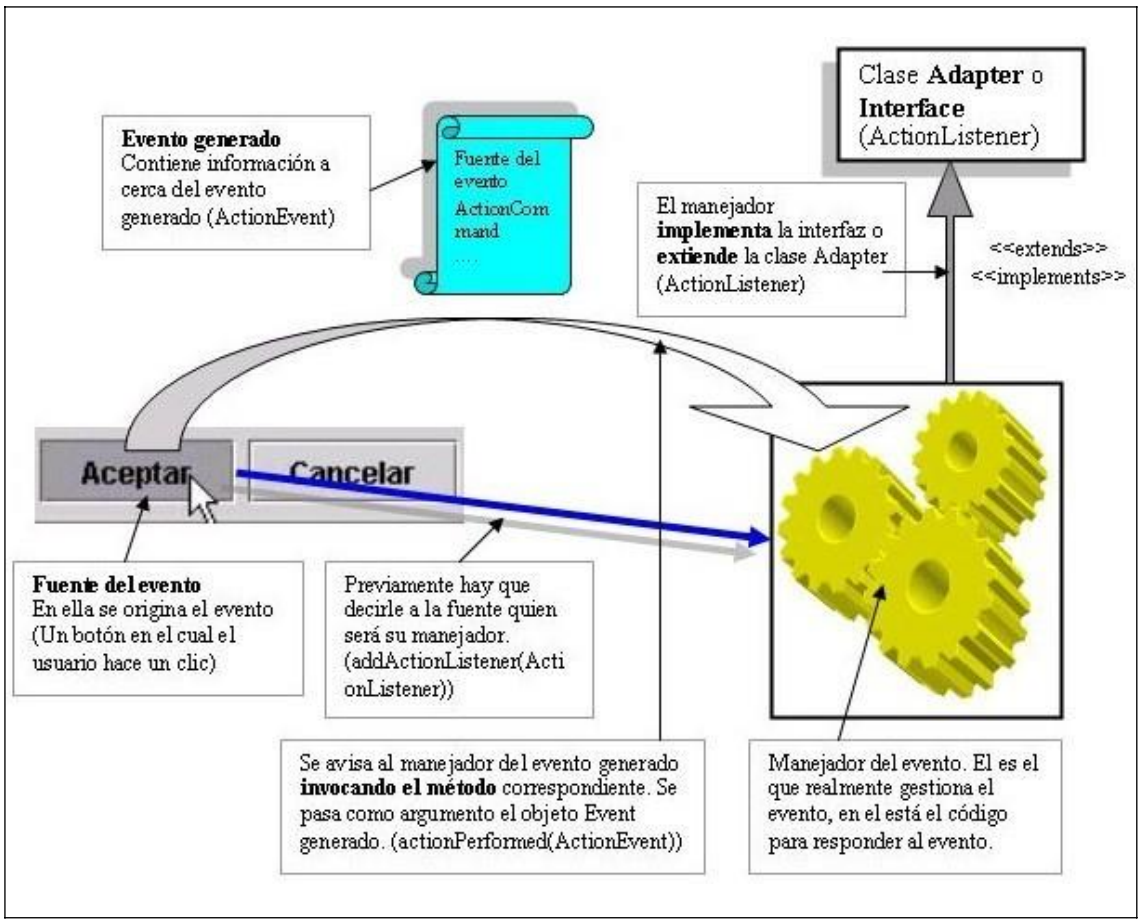
2.2 Eventos

El sistema de gestión de eventos de Java 1.2 es el mismo que Java 1.1 y por lo tanto el mismo que para las librerías AWT. Aunque los desarrolladores de Java considerasen que para mejorar el lenguaje se necesitaba dejar a un lado las librerías AWT e introducir las Swing no sintieron lo mismo del sistema de gestión de eventos, consideraron que era lo suficientemente bueno.

Realmente este sistema de gestión de eventos es bastante elegante y sencillo, sobre todo si se compara con el sistema de gestión de eventos de Java 1.0, mucho más engorroso de usar y menos elegante.

Todos los sistemas operativos están constantemente atendiendo a los eventos generados por los usuarios. Estos eventos pueden ser pulsar una tecla, mover el ratón, hacer clic con el ratón, pulsar el ratón sobre un botón o menú (Java distingue entre simplemente pulsar el ratón en un sitio cualquiera o hacerlo, por ejemplo, en un botón). El sistema operativo notifica a las aplicaciones que están ocurriendo estos eventos, y ellas deciden si han de responder o no de algún modo a este evento.

El modelo de Java se basa en la **delegación de eventos**: el evento se produce en un determinado componente, por ejemplo un scroll. Dónde se produce el evento se denomina “fuente del evento”. A continuación el evento se transmite a un “manejador de eventos” (event listener) que este asignado al componente en el que se produjo el evento. El objeto que escucha los eventos es el que se encargará de responder a ellos adecuadamente. Esta separación de código entre generación del evento y actuación respecto a él facilita la labor del programador y da una mayor claridad a los códigos.



Gestión de eventos en Java. A la *fuente* del evento, en este caso un botón, le indicamos quién será su *manejador* de eventos, manejador que ha de *extender la clase Adapter* correspondiente o implementar *la interfaz Listener* (interfaz ActionLitener en este caso). Cuando el usuario genere el evento deseado (en este caso pulse el botón), el objeto fuente empaqueta información acerca del evento generando un objeto de tipo Event (ActionEvent en este caso) e invoca el método correspondiente del manejador (actionPerformed(actionEvent)) pasándole como información el objeto de tipo Event generado. Es responsabilidad del manejador, y no de la fuente, responder al evento, por ello se dice que la fuente delega la gestión del evento en el manejador.

Lo que la fuente de eventos le pasa al objeto encargado de escuchar los eventos es, como no, otro objeto. Es un objeto tipo Event. En este objeto va toda la información necesaria para la correcta gestión del evento por parte del objeto que escucha los eventos.

El objeto que escucha los eventos ha de implementar para ello una **interface**. El nombre de esta **interface** es siempre el nombre del evento más "Listener": para que un objeto escuche eventos de ratón ha de implementar la interface MouseListener, para que escuche eventos de teclado KeyListener...

Para hacer que un objeto escuche los eventos de otro objeto se emplea el método **add[nombre_evento]Listener**, así si tuviésemos un **Jframe** llamado "**frame**" y quisiésemos que el objeto llamado "manejador" escuchase los eventos de ratón de "**frame**" lo haríamos del siguiente modo:

```
frame.addMouseListener(manejador);
```

manejador ha de pertenecer a una clase que implemente la interface MouseListener, que tiene un total de 7 métodos que ha de implementar.

En la siguiente tabla mostramos los eventos más comunes, junto a la interface que debe implementar el objeto que escuche esos eventos y el método para asociar un objeto para escuchar dichos eventos. En la columna de la derecha se presentarán diversos componentes que pueden generar dichos eventos.

Event, listener interface y métodos para ligar el objeto que escucha	**Algunos componentes que generan este tipo de eventos.**
ActionEvent ActionListener addActionListener( )	JButton, JList, JTextField, JmenuItem, JCheckBoxMenuItem, JMenu, JpopupMenu.
AdjustmentEvent AdjustmentListener addAdjustmentListener( )	JScrollbar y cualquier objeto que implemente la interface Adjustable.
ComponentEvent ComponentListener addComponentListener( )	Component, JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
ContainerEvent ContainerListener addContainerListener( )	Container ,JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame.
FocusEvent FocusListener addFocusListener( )	Component , JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
KeyEvent KeyListener addKeyListener( )	Component , JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
MouseEvent MouseListener addMouseListener( )	Component , JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
MouseEvent	Component , JButton, JCanvas,

Event, listener interface y métodos para ligar el objeto que escucha	**Algunos componentes que generan este tipo de eventos.**
MouseMotionListener addMouseMotionListener( )	JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, JTextField.
WindowEvent WindowListener addWindowListener( )	Window, JDialog, JFileDialog, and JFrame.
ItemEvent ItemListener addItemListener( )	JCheckBox, JCheckBoxMenuItem, JComboBox, Jlist.
TextEvent TextListener addTextListener( )	JTextComponent, JtextArea, JTextField.

Cabe preguntarse ahora qué métodos tiene cada interface, ya que hemos de implementar todos ellos, incluso aunque no los usemos, sino la clase que se encargaría de escuchar los eventos sería abstracta y no podríamos crear ningún objeto de ella. Parece un poco estúpido implementar métodos que no hagan nada sólo porque la interface de la que heredamos los tenga. Así por ejemplo si estamos interesados en escuchar clics de ratón hemos de crear una clase que implemente MouseListener, pero nosotros sólo estaremos interesados en un método de dicha interface: mouseClicked.

Los creadores de Java también pensaron en esto y por ello para cada interface que tiene más de un método crearon una clase llamada [nombre_evento]Adapter (MouseAdapter), que lo que hace es implementar todos los métodos de la interface sin hacer nada en ellos. Nosotros lo único que tendremos que hacer es que nuestra clase que escuche eventos extienda esta clase y sobrescriba los métodos que nos interesen.

A continuación en la siguiente tabla damos un listado de las principales interfaces junto a sus respectivas clases **"Adapter"** y los métodos que poseen:

Listener interface y Adapter	**Metodos**
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent)

Listener interface y Adapter	Metodos
	componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Un frame que se cierra

Pongamos en práctica lo que hemos visto haciendo un **frame** que se pueda cerrar:

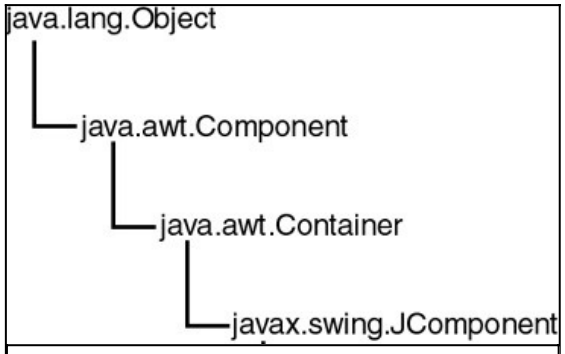
```
import javax.swing.*;
import java.awt.event.*;

class Frame2 extends JFrame {

    public Frame2() {
        setTitle("Hola!!!");
        setSize(300, 200);
        addWindowListener(new manejador());
    }
}

class manejador implements WindowListener {

    @Override
```

```
    public void windowClosing(WindowEvent e) {
        System.out.println("Terminando la aplicación");
        System.exit(0);
    }

    @Override
    public void windowOpened(WindowEvent e) {
        System.out.println("Abriendo la ventana");
    }

    @Override
    public void windowClosed(WindowEvent e) {
             System.out.println("Cerrando la ventana a través de
dispose");
    }

    @Override
    public void windowActivated(WindowEvent e) {
        System.out.println("Ventana activada");
    }

    @Override
    public void windowDeactivated(WindowEvent e) {
        System.out.println("Ventana desactivada");
    }

    @Override
    public void windowIconified(WindowEvent e) {
        System.out.println("Ventana hecha un icono");
    }

    @Override
    public void windowDeiconified(WindowEvent e) {
        System.out.println("Ventana maximizada");
    }
}

public class Ejemplo2 {

    public static void main(String[] args) {
        JFrame frame = new Frame2();
```

```
            frame.setVisible(true);
        }
    }
```

Aquí se ve como al hacer que la clase manejador implemente la interface WindowListener hemos de implementar sus siete métodos, aunque sólo nos interesa el que está relacionado con el cierre de la ventana (no obstante, con fines pedagógicos, aquí hemos escrito código en todos los métodos mostrando un mensaje por pantalla cuando se invocan. A continuación, rescribimos el ejemplo aprovechando las ventajas de las clases **Adapter**:

```
package ejemplosgui;

import javax.swing.*;
import java.awt.event.*;

class Frame3 extends JFrame {
    public Frame3() {
        setTitle("Hola!!!");
        setSize(300, 200);
        addWindowListener(new manejador3());
    }
}

class manejador3 extends WindowAdapter {
    @Override
    public void windowClosing(WindowEvent e) {
        System.out.println("sali");
        System.exit(0);
    }
}
...
```



Jerarquía de JPanel

2.3 JPanel

Ahora ya sabemos hacer una ventana (frame) que se cierra. Podríamos empezar

a añadirle botones, scrolls, campos de texto ... y todo lo que necesitemos, pero no es considerado una buena práctica de programación añadir componentes directamente sobre un contenedor "pesado" (frames por lo que a nosotros respecta). Lo correcto es añadir a este uno o varios paneles y añadir sobre los paneles lo que necesitemos.

Una de las ventajas de añadir paneles sobre nuestro frame es que los paneles al derivar de JComponent poseen el método paintComponent que permite dibujar y escribir texto sobre el panel de modo sencillo.

Para añadir un JPanel a nuestro **frame** primero obtenemos uno de los objetos que forman el **frame**: el "panel contenedor" (content pane). Para ello invocaremos al método **getContentPane** de nuestro **JFrame**. El objeto que nos devuelve será de tipo Container:

```
Container [nombre_del_contentpane] = frame.getContentPane();
```

A continuación invocamos al método **add** del Container obtenido para añadir el panel, pasándole el propio panel al método:

```
[nombre_del_contentpane].add(nombre_del_panel);
```

Añadámosle un JPanel a nuestro frame (solo se muestra el código del constructor; el resto del código es idéntico al del ejemplo anterior):

```
    public Frame4() {
        setTitle("Hola!!!");
        setSize(300, 200);
        addWindowListener(new manejador4());
        Container contentpane = getContentPane();
        JPanel panel = new JPanel();
        contentpane.add(panel);
        panel.setBackground(Color.red);

    }
```

2.4 Layouts

Ya ha llegado casi el momento de empezar a añadir cosas a nuestra ventana, sólo una cosa queda pendiente: como controlar dónde añadimos los objetos. Por ejemplo, cómo decirle a nuestro panel dónde tiene que colocar un botón.

Una solución sería indicarle dónde colocar la esquina izquierda de arriba del botón y luego indicar el alto y ancho del botón. Esto es precisamente lo que hace el método setBounds(int,int,int,int) de la clase Component. Parece, en principio, una buena solución. Hagámoslo:

```
public Frame5() {
        setTitle("Hola!!!");
        setSize(500, 400);
        addWindowListener(new manejador5());
        Container contentpane = getContentPane();
        JPanel panel = new JPanel();
        panel.setLayout(null);
        JButton boton = new JButton();
        boton.setBounds(300, 200, 50, 90);
        panel.add(boton);
        contentpane.add(panel);
        panel.setBackground(Color.red);
    }
```

Si con este código probamos a cambiar de tamaño la ventana podremos ver como el botón no se mueve, desapareciendo si la ventana se hace muy pequeña y cambiando su posición relativa a los bordes de la ventana. Esto, en una aplicación real, desorientaría al usuario que nunca sabría dónde irlo a buscar al cambiar el tamaño de la ventana.

Para solucionar este inconveniente se crearon los Layout Maneger: con ellos se especifican unas posiciones determinadas en un panel, frame o applet donde añadiremos nuestros componentes o un nuevo panel, al que también le podremos añadir un layout en cuyas posiciones podremos añadir componentes o más panels con layouts....

La posibilidad de añadir varios panels a un layout y fijar a éstos nuevos layouts da un gran dinamismo a la hora de colocar los componentes.

FlowLayout

Es el que tienen los paneles por defecto. Los objetos se van colocando en filas en el mismo orden en que se añadieron al contenedor. Cuando se llena una fila se pasa a la siguiente. Tiene tres posibles constructores:

```
FlowLayout();
```

Crea el layout sin añadirle los componentes, con los bordes de unos pegados a otros.

```
FlowLayout(FlowLayout.LEFT[RIGTH][CENTER]);
```

Indica la alineación de los componentes: a la izquierda, derecha o centro.

```
FlowLayout(FlowLayout.LEFT, gap_horizontal, gap_vertical);
```

Además de la alineación de los componentes indica un espaciado (gap) entre los distintos componentes, de tal modo que no aparecen unos pegados a otros. He aquí un ejemplo del uso de FlowLayout:

```
package ejemplosgui;

import javax.swing.*;
import java.awt.*;


 class Frame6 extends JFrame{

    public  Frame6() {
       setSize(600,400);
       Container container  =  this.getContentPane();
       FlowLayout fl = new FlowLayout(FlowLayout.LEFT, 5, 10);
       container.setLayout(fl);
       for (int i=0; i<14; i++) {
          JButton button = new JButton("Button"+(i+1));
```

```
            button.setPreferredSize(new Dimension(100+(i%5)*20,25+(i
%4)*20));
          container.add(button);
       }
    }
}

public class Ejemplo6{
    public static void main (String[] args){
        Frame6 frame = new Frame6();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

GridLayout

Como su propio nombre indica crea un **grid (malla)** y va añadiendo los componentes a las cuadrículas de la malla de izquierda a derecha y de arriba abajo. Todas las cuadrículas serán del mismo tamaño y crecerán o se harán más pequeñas hasta ocupar toda el área del contenedor. Hay dos posibles constructores:

```
   GridLayout(int filas, int columnas);
```

Creará un **layout** en forma de malla con un número de columnas y filas igual al especificado.

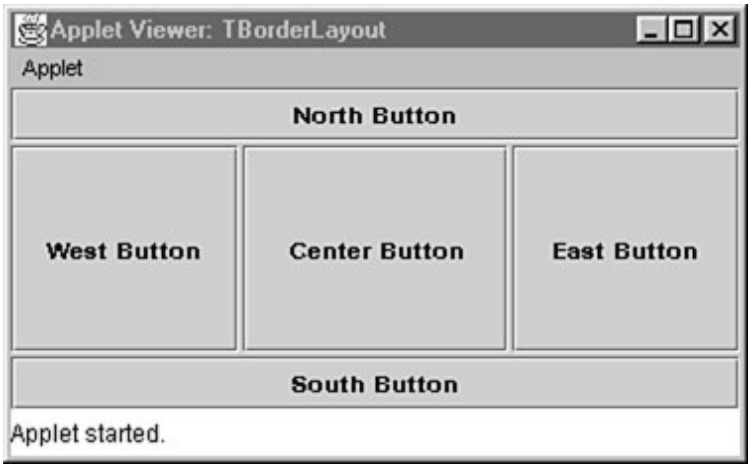
```
GridLayout(int columnas, int filas, int gap_horizontal, int
gat_vertical);
```

Especifica espaciados verticales y horizontales entre las cuadrículas. El espaciado se mide en píxeles. Veamos un ejemplo (solo se muestra el constructor del frame del ejemplo; el resto del código es idéntico al ejemplo anterior):

```
    public Frame7() {
        setSize(350, 150);
```

```
        Container container = this.getContentPane();
        GridLayout grid = new GridLayout(3, 2, 5, 5);
        container.setLayout(grid);

        for (int i = 0; i < 6; i++) {
            container.add(new JButton("Button" + (i + 1)));
        }
    }} ///:~
```

BorderLayout



Este layout tiene cinco zonas predeterminadas: son **norte (NORTH), sur (SOUTH), este (EAST), oeste (WEST) y centro (CENTER)**. Las zonas norte y sur al cambiar el tamaño del contenedor se estirarán hacia los lados para llegar a ocupar toda el área disponible, pero sin variar su tamaño en la dirección vertical. Las zonas este y oeste presentan el comportamiento contrario: variarán su tamaño en la dirección vertical, pero sin variarlo en la dirección horizontal. En cuanto a la zona central crecerá o disminuirá en todas las direcciones para rellenar todo el espacio vertical y horizontal que queda entre las zonas norte, sur, este y oeste. Posee dos constructores:

```
    BorderLayout();
```

que creará el layout sin más y

```
    BordreLayout(int gap_horizontal, int gap_vertical);
```

que creará el layout dejando los gaps horizontales y verticales entre sus distintas zonas.

A la hora de añadir más paneles o componentes a este Layout hay una pequeña diferencia respecto a los otros dos: en los otros íbamos añadiendo componentes y él los iba situando en un determinado orden, aquí especificamos en el método add la región donde queremos añadir el componente:

```
    panel.add(componente_a_añadir, BorderLayout.NORTH);
```

Con esta llamada al método add añadiremos el componente en la región norte. Cambiando NORTH por SOUTH, EAST, WEST, CENTER lo añadiremos en la región correspondiente. Veamos un ejemplo (nuevamente, sólo se muestra el constructor):

```
public  Frame8() {
        setSize(400,250);
       Container container = this.getContentPane();
        container.setLayout(new BorderLayout(2,2));

       String[] borderConsts = { BorderLayout.NORTH,
                                 BorderLayout.SOUTH,
                                 BorderLayout.EAST,
                                 BorderLayout.WEST,
                                 BorderLayout.CENTER };

       String[] buttonNames = { "North Button", "South Button",
                                "East Button", "West Button",
                                "Center Button" };

       for (int i=0; i<borderConsts.length; i++) {
          JButton button = new JButton(buttonNames[i]);
          container.add(button, borderConsts[i]);
       }
}
```

2.5 JButton

Ha llegado el momento de introducir un componente que no sea un mero contenedor. Empecemos por un botón. Crear un botón es tan sencillo como:

```
    JButton boton = new JButton();
```

Si queremos que el botón aparezca con una etiqueta de texto:

```
JButton boton = new JButton("texto va aquí");
```

Veamos como añadirle funcionalidad a nuestro botón. Cada vez que hacemos clic sobre el botón se genera un evento del tipo **ActionEvent**. Para poder escuchar dichos eventos necesitaremos una clase que implemente la interface **ActionListener**, interface que tiene un solo método **actionPerformed** (ActionEvent).

Haremos, por ejemplo, que al hacer un clic sobre nuestro botón cambie el color de fondo de un panel. Reutilizaremos nuestro Ejemplo5. Le pondremos un BorderLayout y haremos que el panel sea quien escuche los eventos del botón; para ello ha de implementar la interface ActionListener. Cuando se llame al método actionPerformed invocaremos al método setBackground del panel para cambiarlo a azul. Por otro lado, hemos de indicar que va a ser el frame el que escuche los eventos del botón:

```
boton.addActionListener(this);
```

El código resultante será:

```
...
class Frame9 extends JFrame implements ActionListener {

    private JPanel panel = new JPanel();
    private JButton azul;

    public Frame9() {
        setTitle("Hola!!!");
        setSize(500, 400);
        addWindowListener(new manejador9());
        Container contentpane = getContentPane();
        panel.setLayout(new BorderLayout());

        azul = new JButton("Azul");
        azul.addActionListener(this);
        Dimension tamano = new Dimension();
        tamano.height = 40;
```

```
        tamano.width = 100;
        azul.setPreferredSize(tamano);
        panel.add(azul, BorderLayout.SOUTH);

        contentpane.add(panel);
        panel.setBackground(Color.red);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        panel.setBackground(Color.blue);
    }
}

class manejador9 extends WindowAdapter {

    @Override
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
...
```

Haremos una aclaración sobre lo que se ha hecho en el ejemplo; se invoca a un método del botón, setPreferredSize(Dimension). Este método fija un par de constantes del objeto botón a los valores indicados por el objeto dimensión, que no es más que un objeto que contiene dos enteros. Estos valores son el tamaño que preferentemente ha de tener el botón. BorderLayout respetará la altura, ya que al añadirlo en la posición sur no crecerá en esta dimensión. Si el botón se hubiese añadido en las posiciones este u oeste, donde no crece la anchura sería este el parámetro que respetaría BorderLayout.

El motivo de haber hecho esto es que en caso de no hacerlo el botón sería muy fino, con lo cual quedaría poco estético y además no se daría leído la etiqueta.

En cuanto al objeto panel no ha sido definido en el constructor porque en caso de haberlo hecho sólo existiría esa variable dentro del constructor y no podríamos invocar al método **setBackgroundColor** en el método **ActionPerformed** por pertenecer a otro bloque distinto que el de inicialización.

Compliquemos un poco más el problema. Añadamos un botón en las posiciones norte, este y oeste y hagamos que según pulsemos un botón u otro cambie a un

color distinto el fondo del panel. Haremos que sea el mismo panel el que escuche los eventos de todos los botones.

El problema que se nos plantea ahora es el siguiente: cada vez que un **botón sea pulsado** se generará un **ActionEvent** y se invocará al método **ActionPerformed**, pero ¿Cómo sabremos que botón fue accionado para saber de qué color ha de tener el fondo del panel? Esto lo lograremos gracias a que en **el objeto evento (ActionEvent) hay información sobre quien produjo dicho evento: invocando al método getSource() de ActionEvent** nos devuelve el nombre del componente que generó dicho evento. Así habremos resuelto nuestro problema:

```
...
class Frame extends JFrame implements ActionListener {

    private JPanel panel = new JPanel();
    private JButton azul, rosa, amarillo, verde;

    public Frame() {
        setTitle("Hola!!!");
        setSize(500, 400);
        addWindowListener(new manejador10());
        Container contentpane = getContentPane();
        panel.setLayout(new BorderLayout());

        azul = new JButton("Azul");
        azul.addActionListener(this);
        Dimension d = new Dimension();
        d.height = 40;
        d.width = 100;
        azul.setPreferredSize(d);

        verde = new JButton("Verde");
        verde.addActionListener(this);
        verde.setPreferredSize(d);

        amarillo = new JButton("Amarillo");
        amarillo.addActionListener(this);
        amarillo.setPreferredSize(d);
```

```
        rosa = new JButton("Rosa");
        rosa.addActionListener(this);
        rosa.setPreferredSize(d);
        panel.add(azul, BorderLayout.SOUTH);
        panel.add(verde, BorderLayout.NORTH);
        panel.add(amarillo, BorderLayout.EAST);
        panel.add(rosa, BorderLayout.WEST);

        contentpane.add(panel);
        panel.setBackground(Color.red);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == azul) {
            panel.setBackground(Color.blue);
        }
        if (source == verde) {
            panel.setBackground(Color.green);
        }
        if (source == amarillo) {
            panel.setBackground(Color.yellow);
        }
        if (source == rosa) {
            panel.setBackground(Color.pink);
        }
    }
}...
```

2.6 Revisión de algunos componentes Swing

En este apartado haremos una rápida revisión de varios de los componentes de la librería Swing más empleados. Consultando su javadoc, pueden encontrase más detalles sobre qué eventos generan, y cómo configurar su apariencia. La gestión de estos eventos es análoga al ejemplo presentado en el listado 3, y siempre sigue el modelo de delegación de eventos.

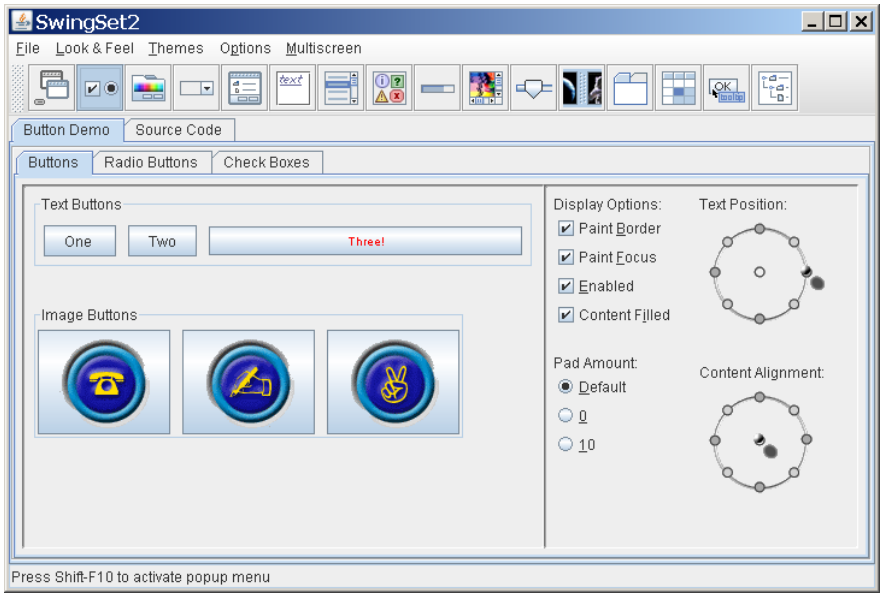
- JTextField: campo de texto pensado para obtener texto del usuario, éste tecleará en él y cuando pulse intro podremos disponer del texto que tecleó. Únicamente se puede recoger una línea de texto. Tiene métodos para recoger el texto del usuario, poner un texto en él, recoger solo el texto seleccionado, seleccionar una parte del texto, insertar texto, cortar texto, pegar texto, etc.
- JTextArea: todo lo dicho para JTextField es válido aquí también; la diferencia entre ambos es que JTextArea permite al usuario introducir más de una línea de texto.
- JPasswordField: campo en el cual al escribir no se ve lo que se escribe, sino un carácter (*, por ejemplo). Se emplea parta pedirle passwords al usuario y evitar que puedan ser leídos por alguien.
- JScrollBar: es el típico scroll que permite desplazarse a lo largo de un componente demasiado grande para mostrarse en pantalla. Puede servir tanto para tomar una entrada numérica del usuario, o para "scrollear" a lo largo de regiones demasiado grandes para ser vistas en la ventana en que representamos la información. Hay un panel de Swing, JScrollPanel, que ya lleva incorporados por defecto dos scrolls, nosotros lo único que tenemos que hacer es introducir en él un componente y él se encargará de gestionar los scrolls horizontales y verticales. JScrollBar posee métodos para fijar el valor numérico correspondiente al mínimo y máximo de las posiciones del scroll, para ver qué valor posee el scroll en un determinado momento, para poner el scroll en un determinado valor, etc.
- JLabel: etiqueta de texto que podemos colocar al lado de cualquier componente para darle una indicación al usuario de cuál es la función de dicho componente. También se puede emplear a modo de título de, por ejemplo, un applet.
- JCheckBox: es un componente empleado para tomar información del usuario sobre cosas del tipo “sí”, “no”; se emplea para seleccionar una opción entre un conjunto de opciones. Posee métodos para seleccionar o deseleccionar, o para indicar su estado.
- JRadioButton: debe su nombre a funcionar como los botones de una radio antigua: al seleccionar uno se deselecciona el que antes estaba seleccionado. Cuando añadimos estos componentes a nuestra interfaz se añaden por grupos; de entre todos los JRadioButtons que han sido añadidos a un grupo sólo puede haber uno seleccionado, y la selección de uno distinto dentro del grupo provoca la inmediata deselección del que antes estaba

seleccionado. Se emplean para darle a elegir al usuario entre un grupo de opciones mutuamente excluyentes.

- JList: componente que permite al usuario seleccionar una opción de una lista, que normalmente lleva un scroll incorporado; la opción se selecciona haciendo clic directamente sobre ella. Se emplea cuando el número de opciones entre las que ha de escoger el usuario es demasiado grande para presentárselas en forma de radiobuttons o checkboxes. Posee métodos para permitir selección simple o múltiple, seleccionar o deseleccionar una opción, averiguar que opción está seleccionada, etc.
- JComboBox: su filosofía es idéntica a la de JList, pero en esta ocasión las opciones no se ven en un principio. El usuario ha de hacer un clic sobre una pestaña que desplegará una lista de opciones sobre las cuales el usuario escoge una mediante un clic.
- JMenu: es el componente que permite generar los típicos menús a los que todos estamos acostumbrados. En estos menús se pueden añadir JChecBoxes y JRadioButtons.

Esta lista está muy lejos de ser una lista exhaustiva de todos los componentes de la librería Swing. La mejor forma de hacerse una idea de lo que es posible hacer con Swing es ejecutando la demo SwingSet2 que viene con todos los jdk. En Windows, puedes encontrar esa demo en el directorio "C:\Archivos de programa\Java\jdk1.8.0_241\demo\jfc\SwingSet2"; con hacer doble clic sobre el archivo "SwingSet2.jar" que encontrarás en esa localización se lanzará la demo que se puede ver en la figura 4. En ella se demuestran todos los componentes de swing, y se muestra el código fuente necesario para conseguir cada uno de los efectos.

En caso de no encontrarla, quizás no tienes instalado el JDK con demos y ejemplos, se puede descargar de https://www.oracle.com/java/technologies/javase-jdk8-downloads.html en la sección de “Java SE Development Kit 8u341 Demos and Samples Downloads” o puedes descargar este fichero del campus virtual.



Otra recomendación que le doy al lector para trabajar con Swing es que emplee el sentido común. Por ejemplo ¿Habrá algún método dentro de JList se me permita cambiar el tipo de fuente que emplea para mostrar los elementos de la lista? Parece que tiene bastante sentido que exista esta posibilidad ¿no? Y efectivamente, existe. Si la intuición dice que un determinado método debería existir en un componente, lo más probable es que exista. Y, habitualmente, con echar un vistazo al javadoc y leer los nombres de los métodos es suficiente para localizarlo.

3 Diseñadores gráficos de interfaces Swing

Cualquier entorno de desarrollo Java actual que se precie incluye un diseñador gráfico de aplicaciones Swing. Estos diseñadores son herramientas en las cuales es posible construir una aplicación Swing simplemente seleccionando componentes de una paleta y arrastrándolos a una ventana en la cual vamos construyendo nuestra interfaz de usuario. Los diseñadores también nos permiten generar de modo automático gran parte del código necesario para la gestión de un evento. Lo más habitual cuando desarrollamos una aplicación de escritorio es apoyarnos en una de estas herramientas, y no escribir todo el código a mano cómo hemos hecho en este artículo. No obstante, para aprender programación gráfica lo más recomendable es hacer lo que hemos hecho en este artículo: escribir el código a mano. Si comenzamos a emplear directamente los diseñadores gráficos, no comprenderemos el código que generan y, cuando este código no se ajuste a nuestras necesidades, nos será imposible modificarlo. También nos será imposible retocar ese código sin la ayuda del diseñador.

4 Dibujando cosas en la pantalla

En este apartado vamos a ver como dibujar sobre una ventana. Esto puede sernos útil tanto para mostrar resultados de un modo gráfico al usuario como para realizar animaciones que ilustren nuestras páginas web. Si recordamos lo que ya se comentó al principio del tema, los componentes Swing están "dibujados" sobre una ventana que le pedimos al sistema operativo, se puede decir que "son de la máquina virtual Java" y no del sistema operativo. Esto no ocurría así con los componentes de la librería AWT, en la cual los componentes "eran del S.O.". Comentamos al principio de este tema que una de las ventajas de este hecho es un mayor control sobre la apariencia de estos componentes...

Si estuviésemos programando con las antiguas librerías AWT sólo hay un componente sobre el cual podemos dibujar: Canvas. Es el único objeto que el S.O. nos proporciona en el que podemos dibujar. Esto quiere decir que, si por ejemplo no nos gusta la apariencia de un scroll o de un botón no podemos hacer nada para modificarla, hemos de aceptarla tal y como se nos da. Por este motivo una misma aplicación se veía diferente en los diferentes S.O., un scroll de Windows no tiene por qué ser igual que un scroll de Linux o Mac. En las nuevas librerías Swing es posible hacer que una misma aplicación se vea igual en todos los S.O., mediante una característica de los componentes que se llama Look and Feel.

Por otra parte, en las librerías Swing es posible pintar sobre cualquier componente que derive de JComponent, es posible por lo tanto modificar la apariencia de un botón o scroll a nuestro antojo (al margen de los distintos Looks and Feel disponibles. De todas formas, no es una buena práctica de programación pintar sobre cualquier componente En la documentación de Java se aconseja que si queremos mostrar al usuario un dibujo que lo hagamos siempre sobre un JPanel.

4.1 Empezando a dibujar

A continuación vamos a ver qué tenemos que hacer para dibujar en Swing. Lo primero es hacerse con el objeto gráfico del componente sobre el cual queremos dibujar. El objeto gráfico es un objeto que posee toda la información que un componente necesita para dibujarse en pantalla. Para obtener el objeto gráfico empleamos el método getGraphics(), de JComponent.

```
Graphics g = Componente.getGraphics()
```

El objeto que nos devuelve este método realmente es una instancia de la clase Graphics2D, una clase que extiende Graphics y que nos proporciona primitivas de pintado más avanzadas. Por ello lo primero que se suele hacer con este objeto es convertirlo a Graphics2D:

```
Graphics2D g2d = (Graphics2D) Componente.getGraphics();
```

Una vez obtenido dicho objeto gráfico procedemos a dibujar empleando los métodos que dicho objeto nos proporciona. Estos métodos son tan intuitivos como numerosos, por lo que no haremos aquí una revisión de ellos. Si deseamos dibujar algo lo más normal es acudir a la documentación de las librerías y buscar en ella los métodos que necesitemos. A modo de ejemplo, si queremos dibujar una línea y vamos a la documentación de la clase Graphics encontraremos un método drawLine(int x1, int x2, int x3 int x4) en el cual (x1, x2) es el punto de inicio de la línea y (x3, x4) el fin de la línea. Veamos un ejemplo de esto.

```
package ejemplosdibujar;

import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

public class Dibujar1 extends JFrame implements
ActionListener {

    JPanel panel = new JPanel();

    public Dibujar1() {
        setSize(400, 400);
        setTitle("Dibujo");
        Container contentpane = this.getContentPane();
        contentpane.setLayout(new BorderLayout());
        JPanel panelSur = new JPanel();
        contentpane.add(panelSur, BorderLayout.SOUTH);
        JButton boton = new JButton("Dibujar");
        boton.addActionListener(this);
        boton.setActionCommand("Dibujar");
```

```
        panelSur.add(boton);
        JButton boton2 = new JButton("Salir");
        boton2.addActionListener(this);
        boton2.setActionCommand("Salir");
        panelSur.add(boton2);
        contentpane.add(panel, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        String comando = e.getActionCommand();
        if (comando.equals("Salir")) {
            System.exit(0);
        }
        Graphics2D g2d = (Graphics2D) panel.getGraphics();
        g2d.setColor(Color.blue);
        g2d.drawLine(0, 0, 100, 100);
                            g2d.drawLine(150,   150,   (int)
(this.getSize()).getWidth(),                          (int)
(this.getSize()).getHeight());
        g2d.drawRect(100, 80, 200, 200);
        g2d.setColor(Color.red);
        g2d.fillRect(110, 90, 150, 150);
    }

    public static void main(String[] args) {
        (new Dibujar1()).setVisible(true);
    }
}
```

Este código sin embargo tiene un pequeño problema. Si, por ejemplo, minimizamos la ventana y la volvemos a maximizar el dibujo ¡desaparece! En el siguiente apartado veremos porqué.

4.2 El método paintComponent

Cuando cualquier componente de las librerías Swing por cualquier razón necesita **"redibujarse"** en pantalla llama al método **paintComponent**. En este método se halla toda la información que se necesita para dibujar el componente en pantalla.

Causas externas a un componente que fuerzan que este se "redibuje" son que la ventana en la que está se minimiza y luego se maximiza o que otra ventana se ponga encima de la ventana que lo contiene. Además el componente se dibujará cuando se crea y cuando se invoque el método **repaint()**.

Cuando este método es invocado lo único que aparecerá en el componente es lo que se dibuje desde él, todo lo demás es "borrado". Este es el motivo por el cual en nuestro ejemplo anterior al minimizar y luego maximizar la pantalla dejamos de ver lo que habíamos dibujado. Si queremos que siempre que el componente se redibuje aparezcan nuestros dibujos hemos de sobrescribir el método paintComponent y escribir en él el código necesario para realizar estos dibujos. Veamos como haríamos esto:

```
package ejemplosdibujar;

import javax.swing.*;
import java.awt.*;

public class Dibujar2 extends JFrame {

    MyPanel panel = new MyPanel();

    public Dibujar2() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 400);
        setTitle("Dibujo");
        Container contentpane = this.getContentPane();
        contentpane.setLayout(new BorderLayout());
        contentpane.add(panel, BorderLayout.CENTER);
    }

    public static void main(String[] args) {
        (new Dibujar2()).setVisible(true);
```

```
    }
}

class MyPanel extends JPanel {

    public void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setColor(Color.blue);
        g2d.drawLine(0, 0, 100, 100);
                            g2d.drawLine(150,   150,   (int)
(this.getSize()).getWidth(),                          (int)
(this.getSize()).getHeight());
        g2d.drawRect(100, 80, 200, 200);
        g2d.setColor(Color.red);
        g2d.fillRect(110, 90, 150, 150);
    }
}
```

Podemos observar como en el método **paintComponent** lo primero que hay es una llamada al método de la clase padre. **Esto es necesario porque el método de la clase padre es el que tiene idea de cómo "dibujar" el componente**, (nosotros nos limitamos a dibujar sobre el componente, pero no a dibujarlo a él) y además realiza ciertas tareas necesarias para el correcto funcionamiento del componente, como el buffering de las operaciones de pintado, de un modo totalmente transparentes al usuario. **Si no invocamos al método del padre la ventana no se dibujará de modo correcto, por lo que la primera llamada debe ser siempre lo primero que hagamos en el método paintComponent.**

Por último, decir que nunca debemos llamar al método paintComponent de modo directo, hemos de llamar siempre a repaint y será él el que se encargue de llamar a paintComponent con el argumento correcto.