FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

CURSOS INSTITUCIONALES

LENGUAJE DE PROGRAMACIÓN JAVA AVANZADO

Del 29 de Agosto al 06 de Septiembre de 2007

APUNTES GENERALES

CI - 184

Instructor: Ing. Alejandro Velázquez Mena
FONHAPO

Agosto/Septiembre de 2007

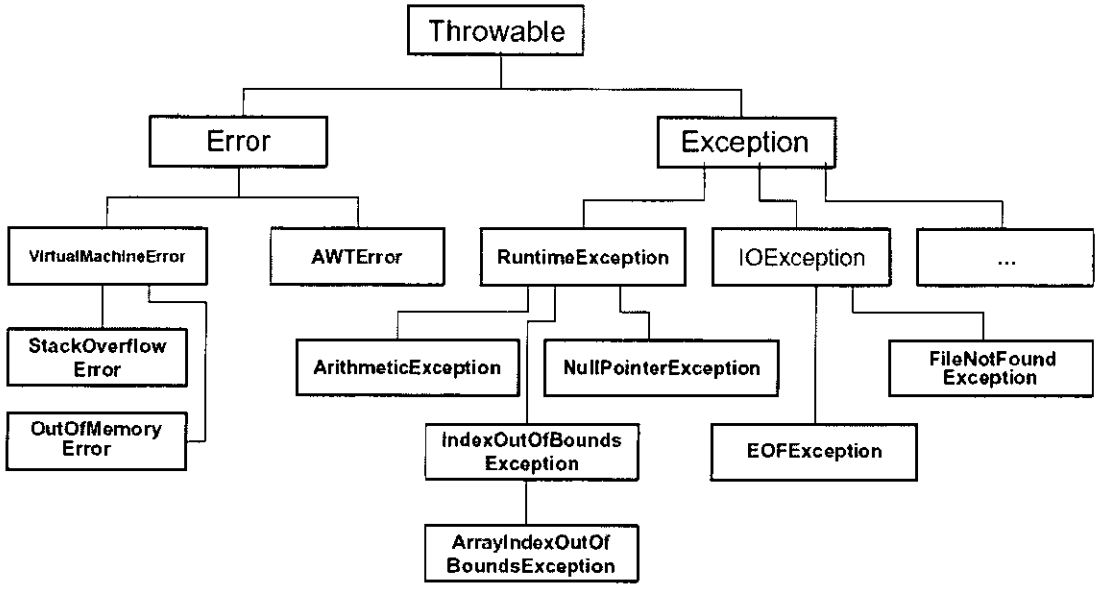
Índice

1. Excepciones.

La detección y manejo de errores es un ingrediente muy importante en la construcción de una aplicación. Java arma a los desarrolladores con un elegante mecanismo de manejo de errores: el manejador de excepciones. El manejador de excepciones permite a los desarrolladores detectar errores fácilmente sin escribir código especial.
Java provee dos categorías de excepciones checked y unchecked. Las excepciones checked son las que el programador espera manejar en el programa. Las excepciones unchecked pueden deberse a bugs o a situaciones consideradas generalmente muy difíciles de manejar.

1.1 Jerarquía excepciones.

Una vez que se han discutido las excepciones como concepto, sabes que éstas son lanzadas cuando un problema ocurre y sabemos que afecta el flujo del programa. Todas las excepciones provienen de la clase Exception, que a su vez deriva de la clase Throwable y ésta deriva de la clase Object.



Como se puede observar, hay dos subclases que derivan de Throwable: Exception y Error. Las clases que derivan de Error representan situaciones inusuales que no son causadas por errores de programación e indican cosas que no normalmente pasan durante la ejecución de un programa. Generalmente las aplicaciones no son capaces de recuperarse de un Error, por tal motivo no se requiere que uno las maneje.

1.2 Cachando una excepción usando try y catch.

El término excepción significa "condición excepcional" y esta condición afecta el flujo del programa. Cuando una excepción ocurre en Java una excepción es lanzada. El código responsable de hacer algo con la excepción es el manejador de excepciones y cacha (catch) la excepción lanzada.

Cuando una excepción ocurre necesitamos decirle a la JVM que código ejecute; para hacer esto se usan las palabras "try" y "catch". El bloque "try" es usado para definir en donde una excepción puede ocurrir; éste bloque es llamado región de garantía. Una o más cláusulas "catch" especifican el tipo de excepciones que ese bloque de código puede manejar.

```
1. try {
2. // En éste bloque se define la región de garantía.
…
…
…
6. }
7. catch(MyFirstException) {
8. // Aquí se coloca el código encargado de manejar la excepción
…
…
…
11. }
```

1.3 Usando finally

A través de "try" y "catch" Java provee un magnífico mecanismo para atrapar y manejar excepciones. El único conflicto que causa el utilizar estas sentencias es que no se limpia el código que se utilizó después de realizar el "try" y para resolver éste problema Java ofrece el bloque "finally". El bloque "finally" tiene código que siempre es ejecutado después del "try" aunque se ejecute o no una excepción.

```
1: try {
2: // Región de garantía
3: }
4: catch(MyFirstException) {
5: // Código manejador de la excepción
6: }
7: catch(MySecondException) {
8: // Código manejador de la excepción
9: }
10: finally {
11: // Código  a ejecutarse siempre.
12: /
13: }
14:
```

A continuación se muestran códigos legales utilizando varienates con try, catch y finally.

```
//----------codigo 1

try {
// código
} finally {
// código
}

//--------codigo 2

try {
// código
} catch (SomeException ex) {
// código
```

```
} finally {
// código
}
```

Los siguientes son códigos ilegales

```
//---------------codigo 1
try {
// código
}
// necesita un catch o finally
System.out.println("out of try block");


//---------------código 2

try {
// do stuff
}
// No puede haber código intermedio
System.out.println("out of try block");
catch(Exception ex) { }
```

1.4 Propagación de excepciones.

En algunos casos cuando se manda una excepción y ésta no es manejada inmediatamente, la excepción es lanzada al método que la ocasionó. Si la excepción tampoco es manejada en éste método se sigue lanzando hasta que llega al método main y si éste tampoco la maneja el programa termina anormalmente.

El siguiente es un ejemplo:

```
class Birds {
 public static void main(String [] args) {
 try {
       throw new Exception();
 } catch (Exception e) {
       try {
            throw new Exception();
       } catch (Exception e2) { System.out.print("inner "); }
       System.out.print("middle ");
 }
       System.out.print("outer ");
 }
}
```

En muchas ocasiones se va a querer manejar tanto superclases de excepciones, como subtipos de las mismas, pero no que sean manejadas por separado, si no, todas juntas. Cuando una excepción es lanzada, la JVM tratará de encontrar un bloque catch para ese tipo de excepciones. Si no encuentra ninguno tratará buscar un supertipo de la excepción, hasta encontrar el más adecuado.

```
1: import java.io.*;
2: public class ReadData {
3: public static void main(String args[]) {
4: try {
5:     RandomAccessFile raf =
6:     new RandomAccessFile("myfile.txt", "r");
7:     byte b[] = new byte[1000];
8:     raf.readFully(b, 0, 1000);
9: }
```

```
10: catch(FileNotFoundException e) {
11:    System.err.println("File not found");
12:    System.err.println(e.getMessage());
13:    e.printStackTrace();
14: }
15: catch(IOException e) {
16:    System.err.println("IO Error");
17:    System.err.println(e.toString());
18:    e.printStackTrace();
19: }
20: }
21: }
```

En éste pequeño programa se esta tratando de abrir un archivo para leer datos. La lectura y escritura de archivos puede generar muchas excepciones de tipo `IOException`, pero en éste programa queremos saber cuando se realice una `FileNotFoundException`. De otra manera no se sabe cuál es exactamente el problema. De esta manera se esta manejando de manera separada `FileNotFoundException` e `IOException`. Si se generara otra excepción como `EOFException`, será manejada por el bloque que tiene la excepción `IOException`. Las subclases de una excepción tienen que colocarse antes que las superclases, de lo contrario no compilarán. El siguiente código no compila:

```
try {
      // do risky IO things
} catch (IOException e) {
      // handle general IOExceptions
} catch (FileNotFoundException ex) {
      // handle just FileNotFoundException
}
```

El anterior código devolverá el siguiente error de compilación:

```
TestEx.java:15: exception java.io.FileNotFoundException has
already been caught
} catch (FileNotFoundException ex) {
  ^
```

1.5 Declaración de excepciones.

Así como un método indica los argumentos que debe tener y el tipo de retorno, de igual manera tiene que especificar las excepciones que lanza. Para especificar las excepciones que un método va a lanzar se utiliza la palabra "throws" de la siguiente manera:

```
void myFunction() throws MyException1, MyException2 {
// code for the method here
}
```

El método anterior tiene void como tipo de retorno, no acepta ningún argumento y declara que puede lanzar dos excepciones, ya sea de tipo `MyException1 o MyException2`.
Suponiendo que el método que el método no lance directamente la excepción, pero manda llamar a un método que si lo hace, se puede evitar hacer el uso del manejo de la excepción y sólo declarar que ese método puede lanzar una excepción.

Cuando se quieren crear las propias excepciones, ésta simplemente debe ser una subclase de Exception de la siguiente manera:

```
public class BadFoodException extends Exception {
```

```
    public BadFoodException() {
        super();
    }

    public BadFoodException(String msj) {
        super(msj);
    }
}
```

Y cuando se lance esta excepción se haría de la siguiente manera:

```
public class MyException {


    public MyException() {
    }

    public void checkFood(String s) throws BadFoodException{

        System.out.println(s);
        throw new BadFoodException("Mensaje de error:"+s);

    }

     public void checkFood1(String s) throws BadFoodException {
        checkFood(s);
     }

    public static void main(String[] a){

        MyException me = new MyException();
        try{
            me.checkFood1("comida");
        }
        catch(BadFoodException bfe){
            bfe.printStackTrace();
            System.out.println("comida");
        }


    }
}
```

La siguiente tabla presenta las principales excepciones que se generan al momento de programar:

Exception (Chapter Location)	Description	Typically Thrown
ArrayIndexOutOfBoundsException (Chapter 3, "Assignments")	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2, "Object Orientation")	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException (Chapter 3, "Assignments")	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException (Chapter 6, "Formatting")	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.	Programmatically
NullPointerException (Chapter 3, "Assignments")	Thrown when attempting to access an object with a reference variable whose current value is `null`.	By the JVM
NumberFormatException (Chapter 6, "Formatting")	Thrown when a method that converts a String to a number receives a String that it cannot convert.	Programmatically
AssertionError (This chapter)	Thrown when a statement's boolean test returns `false`.	Programmatically
ExceptionInInitializerError (Chapter 3, "Assignments")	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (This chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError (Chapter 10, "Development")	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

2 Entrada y Salida (java.io)

Lo siguiente es un resumen de las clases más importantes que se encuentran el API de java.io.

File: Esta es una representación abstracta de un archivo o un directorio. Esta clase se utiliza para crear archivos, borrar archivos, hacer directorios, trabajan con paths.

FileReader: Esta clase se utiliza para leer caracteres simples de archivos.

BufferReader: Esta clase utiliza clases que heredan de Reader y hacen más eficiente la lectura de archivos, extrae mayor cantidad de datos y los almacena en un buffer, esto minimiza el número de acceso al archivo.

FileWriter: Esta clase es utilizada para escribir caracteres a un archivo.

BufferedWriter: Esta clase es más eficiente que FileReader, ya que minimiza el número de accesos al archivo.

PrintWriter: Esta clase es más flexible que cualquier otro Writer, y no necesariamente necesita un objeto que hereda de la clase Writer.

2.1 Creando archivos con File.

El siguiente ejemplo crea un archivo, hay que notar que cuando se crea un objeto de la clase File, no se está creando ningún archivo o directorio. Esto se hace hasta que se utiliza el método `createNewFile()` o `mkdir()`.

```
import java.io.*;
class Writer1 {
      public static void main(String [] args) {
            try { // warning: exceptions possible
            boolean newFile = false;
            File file = new File("fileWrite1.txt");// it's only an object

            System.out.println(file.exists());      // look for a real
            file
            newFile = file.createNewFile();         // maybe create a
            file!
            System.out.println(newFile);            // already there?
            System.out.println(file.exists());      // look again
            } catch(IOException e) { }
      }
}
```

La salida del código es:

```
false
true
trae
```

2.2 Usando FileWriter y FileReader.

En la práctica no son muy usadas estas clases, pero se muestra el siguiente código para haer uso de éstas clases.

```
import java.io.*;

class Writer2 {
      public static void main(String [] args) {
            char[] in = new char[50]; // to store input
            int size = 0;
            try {
                  File file = new File("fileWrite2.txt"); // just an object
                  FileWriter fw =
                  new FileWriter(file);            // create an actual file
                                                   // & a FileWriter obj
                  fw.write("howdy\nfolks\n");      // write characters to
                                                   // the file
                  fw.close();                      // close file when done
                  FileReader fr =
                  new FileReader(file);            // create a FileReader
                                                   // object
                  size = fr.read(in);              // read the whole file!
                  System.out.print(size + " ");    // how many bytes read
                  for(char c : in)                 // print the array
                        System.out.print(c);
                  fr.close();                      // again, always close
            } catch(IOException e) { }
      }
}
```

La salida del código es:

```
12 howdy
folks
```

2.3 Usando File, Filewriter y PrintWriter.

En el siguiente código se ejemplifica el uso de las clases FileWriter y PrintWriter.

```
File file = new File("fileWrite2.txt"); // create a File object
FileWriter fw = new FileWriter(file);   // create a FileWriter
                                        // that will send its
                                        // output to a File
PrintWriter pw = new PrintWriter(fw);   // create a PrintWriter
                                        // that will send its
                                        // output to a Writer
pw.println("howdy");                    // write the data
pw.println("folks");
```

2.4 Usando File, FileReader y BufferedReader.

En el siguiente código se ejemplifica el uso de las clases FileReader y BufferedReader.

```
File file =
new File("fileWrite2.txt");       // create a File object AND
                                  // open "fileWrite2.txt"
FileReader fr =
new FileReader(file);             // create a FileReader to get
                                  // data from 'file'
BufferedReader br =
new BufferedReader(fr);           // create a BufferReader to
                                  // get its data from a Reader
String data = br.readLine();      // read some data
```

2.5 Combinación de I/O clases (wrapping).

En la siguiente tabla se está un pequeño API de la clase java.io, se muestran los constructores, herencias y métodos de las clases más usadas de éste API.

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()

2.6 Serialización.

Si necesitaras salvar el estado de un objeto, necesitarías guardar cada variable de instancia del objeto, de todos los objetos que necesitas salvar. La peor parte sería tratar de restaurar

el objeto. La serialización ayuda a hacer todo esto, salvar el objeto y todas sus variables de instancia.

La serialización se logra se logra a través de dos métodos.

```
ObjectOutputStream.writeObject() // serialize y escribe
ObjectInputStream.readObject()   // lee y deserializa
```

En el siguiente ejemplo se muestra la serialización de un objeto Cat.

```
class Cat implements Serializable { }                    // 1

public class SerializeCat {
      public static void main(String[] args) {
            Cat c = new Cat();                           // 2
            try {
                  FileOutputStream fs = new FileOutputStream("testSer.ser");
                        ObjectOutputStream os = new ObjectOutputStream(fs);
                        os.writeObject(c);               // 3
                        os.close();
            } catch (Exception e) { e.printStackTrace(); }
            try {
                        FileInputStream          fis          =          new
                        FileInputStream("testSer.ser");
                        ObjectInputStream ois = new ObjectInputStream(fis);
                        c = (Cat) ois.readObject();      // 4
                        ois.close();
            } catch (Exception e) { e.printStackTrace(); }
      }
}
```

1. Se declara la clase Cat que implementa la interfaz Serializable, que no tiene ningún método que implementar.
2. Se crea un objeto de la clase Cat.
3. Se serializa el objeto Cat invocando el método writeObject(). Se serializa el objeto y se escribe dentro de un archivo.
4. Aquí se deserializa el objeto invocando el método readObject(). Éste objeto regresa un objeto y lo deserializa.

¿Qué sucede cuándo se tiene referencias a otras clases? La JVM no guarda estas referencias.

```
class Dog implements Serializable{
      private Collar theCollar;
      private int dogSize;

      public Dog(Collar collar, int size) {
      theCollar = collar;
      dogSize = size;
      }
      public Collar getCollar() { return theCollar; }
}

class Collar {
      private int collarSize;
      public Collar(int size) { collarSize = size; }
      public int getCollarSize() { return collarSize; }
}


import java.io.*;
public class SerializeDog {
      public static void main(String[] args) {
            Collar c = new Collar(3);
```

```
            Dog d = new Dog(c, 8);
            System.out.println("before: collar size is "
            + d.getCollar().getCollarSize());

            try {
                  FileOutputStream fs = new FileOutputStream("testSer.ser");
                  ObjectOutputStream os = new ObjectOutputStream(fs);
                  os.writeObject(d);
                  os.close();
            } catch (Exception e) { e.printStackTrace(); }
            try {
                  FileInputStream fis = new FileInputStream("testSer.ser");
                  ObjectInputStream ois = new ObjectInputStream(fis);
                  d = (Dog) ois.readObject();
                  ois.close();
            } catch (Exception e) { e.printStackTrace(); }

            System.out.println("after: collar size is "
            + d.getCollar().getCollarSize());


      }
}
```

La salida al código anterior es:

```
java.io.NotSerializableException: Collar
```

La excepción indica que la clase Collar también tiene que ser serializada, si modificamos la clase Collar:

```
class Collar implements Serializable {
// mismo código
}
```

La salida del código sería:

```
before: collar size is 3
after: collar size is 3
```

Pero, qué pasa si no se tiene acceso a la clase para serializarla? No siempre se puede heredar, ya que la clase podría ser final o podría tener referencias hacia otros objetos. Para estos casos se cuenta con el modificador "transient", en este caso la serialización se saltará el objeto Collar durante la serialización.

```
class Dog implements Serializable {
      private transient Collar theCollar; // agregando transient
                                          // mismo código
}

class Collar {
    private int collarSize;
    private String collarColor;
    public Collar(int size,String color) {
        collarSize = size;
        collarColor = color;
    }
    public int getCollarSize() {
        return collarSize;
    }

    public String getCollarColor() {
        return collarColor;
    }
}
```

La salida del código es:

```
before: collar size is 3
Exception in thread "main" java.lang.NullPointerException
```

Considerando estos problemas se pueden utilizar los siguientes métodos (siempre deben tener la misma firma):

```
private void writeObject(ObjectOutputStream os) {
      // your code for saving the Collar variables
}
private void readObject(ObjectInputStream os) {
      // your code to read the Collar state, create a new Collar,
      // and assign it to the Dog
}
```

De tal manera que la clase Dog quedaría de la siguiente manera:

```
import java.io.*;

class Dog extends Animal implements Serializable {
    transient private Collar theCollar;
    private int dogSize;
    public Dog(Collar collar, int size) {
        theCollar = collar;
        dogSize = size;

        System.out.println("En dog");
    }

    public Collar getCollar() { return theCollar; }

    private void writeObject(ObjectOutputStream os) {// throws IOException {
      try {
        os.defaultWriteObject();                                    // 1
        os.writeInt(theCollar.getCollarSize());                     // 2
        os.writeObject(theCollar.getCollarColor());                 // 2
      } catch (Exception e) { e.printStackTrace(); }
    }

    private void readObject(ObjectInputStream is) {
    // throws IOException, ClassNotFoundException {
      try {
        is.defaultReadObject();                                     // 3
        theCollar = new Collar(is.readInt(), (String)is.readObject()); // 4
      } catch (Exception e) { e.printStackTrace(); }
     }

}
```

1. Se invoca a defaultwriterObject(), dentro de writeObject(), con esto se le dice a la JVM que haga la serialización de forma natural.
2. Se van a guardar un entero y un String dentro de la serialización.
3. Se va a deserializar el objeto.
4. Se va a crear un nuevo Collar con el entero y el string que guardamos como dato del objeto Collar que no se podía serializar.

En la herencia, si la superclase es serializables, entonces todas las clases hijas lo son también. En el caso de que sea una subclase la que esté serializada, la superclase regresará a su estado inicial por default, esto es porque el constructor de clase vuelve a correr.

```
import java.io.*;
class SuperNotSerial {
	public static void main(String [] args) {
		Dog d = new Dog(35, "Fido");
		System.out.println("before: " + d.name + " "+ d.weight);

		try {
			FileOutputStream fs = new FileOutputStream("testSer.ser");
			ObjectOutputStream os = new ObjectOutputStream(fs);
			os.writeObject(d);
			os.close();
		} catch (Exception e) { e.printStackTrace(); }
		try {
			FileInputStream fis = new FileInputStream("testSer.ser");
			ObjectInputStream ois = new ObjectInputStream(fis);
			d = (Dog) ois.readObject();
			ois.close();
		} catch (Exception e) { e.printStackTrace(); }

		System.out.println("after: " + d.name + " "+ d.weight);
	}
}

class Dog extends Animal implements Serializable {
	String name;
	Dog(int w, String n) {
		weight = w; // inherited
		name = n; // not inherited
	}
}

class Animal { // not serializable !

	int weight = 42;
}
```

La salida del código anterior es:

```
before: Fido 35
after: Fido 42
```

Finalmente las variables estáticas no se pueden serializar, ya que nunca son guardadas como parte del objeto.

3 Threads

Los objetos nos proveen de una manera de dividir un programa en secciones independientes, ya que frecuentemente es necesario separar un programa en partes independientes que sean ejecutables, cada una de estas partes es conocida como un thread (hilo o tarea) y se programan como si cada thread se ejecutara a sí mismo como si contara con el CPU sólo para él. Lo que sucede realmente es que un mecanismo fundamental esta dividiendo el tiempo del CPU por ti, pero generalmente no es necesario que se piense en esto, lo que hace que la programación con múltiples threads sea un aspecto mucho más fácil.

Un proceso es un programa auto ejecutable con su propia dirección de memoria. Un sistema operativo multitareas es capaz de ejecutar más de un proceso (programa) al mismo tiempo, asignando periódicamente ciclos de CPU a cada uno.

Un thread es un control de flujo secuencial y simple dentro de un proceso. Un proceso simple puede entonces tener múltiples threads ejecutables concurrentes.

Un thread está compuesto por tres partes:

- Un CPU virtual.
- El código que ejecuta la CPU.
- Los datos con que trabaja el código.

3.1 Haciendo un thread

Un thread en Java comienza con una instancia de java.lang.Thread. Puedes encontrar métodos referentes a la creación, inicialización y pausa de ellos.

- start()
- yield()
- sleep()
- run()

La acción de un thread comienza en el método run() y la sintaxis que sigue éste método es la siguiente:

```
public void run(){
        // código que se ejecutara
}
```

El código que se necesita ejecutar en un thread separado se escribe en el método run().

Hay dos formas de crear un thread:

- Heredando la clase java.lang.Thread
- Implementando la interfaz Runnable

La versión más usada es la implementación de la interfaz, esto es porque solo existe la herencia simple en Java y por lo tanto sólo se podría heredar una vez.

3.2 Heredando un thread.

Para definir un thread, como ya se dijo se tiene que implementar el método run(), tanto si se hereda de la clase o se implementa de la interfaz.

En el siguiente ejemplo se muestra la creación de un thread heredando de la clase.

```
class MyThread extends Thread {
        public void run() {
                System.out.println("Important job running in MyThread");
        }
}
```

3.3 Implementando java.lang.Runnable

Si se implementa esta interfaz ya se es libre de poder heredar cualquier otra clase.

```
class MyRunnable implements Runnable {
        public void run() {
                System.out.println("Important job running in MyRunnable");
        }
}
```

3.4 Instanciando un thread.

Si se heredo de la clase java.lang.Thread, sólo se tiene que crear una instancia de la clase.

```
MyThread t = new MyThread()
```

Si se implementó la interfaz se tiene que crear una instancia de la clase, luego crear una instancia de Thread y pasarle como argumento la clse Runnable.

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

Si se crea un thread sin argumentos, se llama a al propio método run() de la clase Thread y no al que implementa la clase Runnable.

Si se crean varias instancias de Thread teniendo como argumento la misma clase que implementa Runnable, todos los threads ejecutarán el mismo trabajo.

```
public class TestThreads {
        public static void main (String [] args) {
                MyRunnable r = new MyRunnable();
                Thread foo = new Thread(r);
                Thread bar = new Thread(r);
                Thread bat = new Thread(r);
        }
}
```

Los más importantes constructores de Thread son los siguientes:

- Thread()
- Thread(Runnable r)
- Thread(Runnable r, String name)
- Thread(String name)

Hasta este punto sólo se han credo instancias de thread, pero en ningún momento se le ha dicho al thread que se ejecute. Cuando esto pasa, se dice que esta en estado "new" .

3.5 *Inicializando un Thread.*

Para que un thread se considere vivo, se ejecuta el método start(). Un therad se considera muerto "dead" cuando se ha ejecutado totalmente el código de run(). Cuando se ejecuta el método stara(), el thread pasa de un estado "new" a "runnable", pero esto no implica que el thread se ejecute inmediatamente.

```
class FooRunnable implements Runnable {
      public void run() {
            for(int x =1; x < 6; x++) {
            System.out.println("Runnable running");
            }
      }
}

public class TestThreads {
      public static void main (String [] args) {
            FooRunnable r = new FooRunnable();
            Thread t = new Thread(r);
            t.start();
      }
}
```

Imprimiendo el código anterior se tiene:

```
% java TestThreads
Runnable running
Runnable running
Runnable running
Runnable running
Runnable running
```

Invocando solo el método run(), no se está inicializando el thread, sólo se esta mandando llamar el método.

```
Runnable r = new Runnable();
r.run();
```

En dado caso de que se tengan varios threads ejecutándose, se tiene el método getName(), y de esta manera se sabe el thread que se esta ejecutando.

```
class NameRunnable implements Runnable {
      public void run() {
            System.out.println("NameRunnable running");
            System.out.println("Run by "+ Thread.currentThread().getName());
```

```
        }
}

public class NameThread {
        public static void main (String [] args) {
                NameRunnable nr = new NameRunnable();
                Thread t = new Thread(nr);
                t.setName("Fred");
                t.start();
        }
}
```

Ejecutando el código anterior se tiene:

```
% java NameThread
NameRunnable running
Run by Fred
```

En el siguiente ejemplo se inicializan múltiples threads.

```
class NameRunnable implements Runnable {
        public void run() {
                for (int x = 1; x <= 3; x++) {
                        System.out.println("Run by "
                        + Thread.currentThread().getName()
                        + ", x is " + x);
                }
        }
}

public class ManyNames {
        public static void main(String [] args) {
                // Make one Runnable
                NameRunnable nr = new NameRunnable();
                Thread one = new Thread(nr);
                Thread two = new Thread(nr);
                Thread three = new Thread(nr);
                one.setName("Fred");
                two.setName("Lucy");
                three.setName("Ricky");
                one.start();
                two.start();
                three.start();
        }
}
```

Ejecutando el código anterior se puede obtener como salida:

```
% java ManyNames
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3
Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
```

Hay que tener claro que una vez que el thread ha sido inicializado, no puede inicializarse de nuevo, si se manda llamar stara() por segunda vez, se obtendrá la excepción IllegalThreadStateException.

Algunos métodos importantes de la clase java.lang.Thread que pueden afectar el estado del thread son los siguientes:

- `public static void sleep(long millis) throws InterruptedException`
- `public static void yield()`
- `public final void join() throws InterruptedException`
- `public final void setPriority(int newPriority)`

Los métodos sleep() y join() tienen versions sobrecargadas.

Algunos métodos importantes de la clase java.lang.Object que pueden afectar el estado del thread son los siguientes:

- `public final void wait() throws InterruptedException`
- `public final void notify()`
- `public final void notifyAll()`

El método wait() tiene tres versiones sobrecargadas.

3.6 Estados de un thread y transiciones.

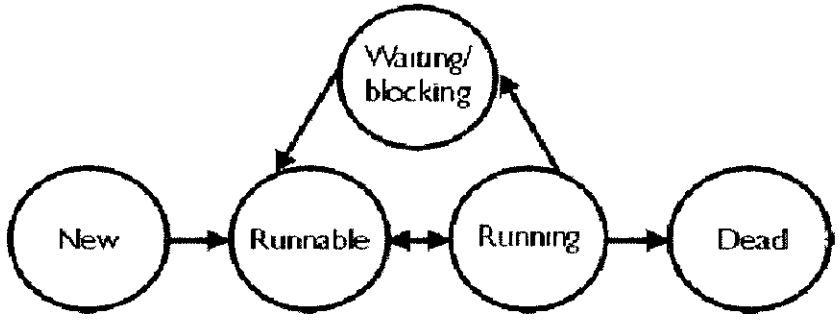
New: Sólo está creada la instancia del thread, pero no se ejecuta; no esta vivo.

Runnable: Es cuando el thread es elejible para correr. Entra en éste estado después de ejecutar start(), en estos momentos se considera vivo.

Running: Es cuando el thread ya se está ejecutando.

Waiting/blocked/sleeping: En este estado el thread sigue vivo, pero no es elegible para ejecutarse, en otras palabras no está en estado runnable, pero puede regresar a ese estado.

Dead: Un thread se encuentra en este estado cuando se ha ejecutado totalmente todo su método run(). Si un thread se muere, no se puede revivir.



Previniendo la ejecución de un thread.

El thread puede estar de tres maneras:

- Sleeping
- Waiting

- Blocked

3.6.1 Método sleep().

Este método es estático, lo que hace es que duerme el thread por un tiempo definido para que después vuelva a ponerse en estado runnable cuando despierte. Este método lanza una InterruptedException.

```
class NameRunnable implements Runnable {
       public void run() {
              for (int x = 1; x < 4; x++) {
                     System.out.println("Run by "
                     + Thread.currentThread().getName());
                     try {
                            Thread.sleep(1000);
                     } catch (InterruptedException ex) { }
              }
       }
}

public class ManyNames {
       public static void main (String [] args) {
              // Make one Runnable
              NameRunnable nr = new NameRunnable();
              Thread one = new Thread(nr);
              one.setName("Fred");
              Thread two = new Thread(nr);
              two.setName("Lucy");
              Thread three = new Thread(nr);
              three.setName("Ricky");
              one.start();
              two.start();
              three.start();
       }
}
```

Ejecutando el código se tiene:

```
% java ManyNames
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
```

3.6.2 Método yield().

Todos los threads son ejecutados con alguna prioridad, es un número entre 1 y 10. La prioridad por default de un thread es la que tiene el thread de ejecución que lo implementa. La clase Thread tiene tres constantes para definir el rango de prioridades:

```
Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5)
Thread.MAX_PRIORITY (10)
```

Mientras mayor sea la prioridad, es llamado más frecuentemente.

El método estático yield() hace que el thread que está en estado running, pase al estado runnable para permitir que otros threads con la misma prioridad tomen su turno, pero no garantiza que se realice. No se recomienda usarse, ya que no todas las JVM tienen implementado este método.

3.6.3 Método join().

Este método hace que un thread se una a otro. El que se une va justo al final del otro y no se va a ejecutar hasta que el thread al que está unido termine su acción.

```
Thread t = new Thread();
t.start();
t.join();
```

En el código anterior se indica: Únete a mí (se lo dice al thread en ejecución, main), al final de t, cuando t termine, podrás pasar a estado runnable de nuevo.

En el siguiente código se estan ejecutando conjuntamente el main y MyThread, esto ahsta que main llega a 50 y se une a MyThread y main vuelve a un estado runnable hasta que MyThread termina su ejecución.

```
class MyThread2 extends Thread {
  public void run() {
         for (int i =0; i < 100; i++)
         System.out.println(" MyThread");
  }

  public static void main(String...args){
           MyThread2 t = new MyThread2();
           t.start();


           for (int i =0; i < 100; i++){

               System.out.println("main");
               if(i == 50){
                 try{
                        t.join();
                 }catch(Exception e){}
               }
           }
  }

}
```

3.7 Sincronización.

La sincronización es un mecanismo en el cual un programador tiene control sobre threads que están compartiendo datos.

En la siguiente clase se tiene una cuenta de la cual se pueden hacer retiros y que comienza con un balance de 50. Ahora hay que imaginar que una pareja Fred y Lucy tienen ambos acceso a esa cuenta. Si los dos vieran la cuenta al mismo tiempo, verían que tienen suficiente dinero como para hacer un retiro, pero no sólo realizan una vez la operación, si no, varias, ¿qué es lo que pasaría?

```
class Account {
	private int balance = 50;
	public int getBalance() {
		return balance;
	}
	public void withdraw(int amount) {
		balance = balance - amount;
	}
}
```

```
public class AccountDanger implements Runnable {
	private Account acct = new Account();
	public static void main (String [] args) {
		AccountDanger r = new AccountDanger();
		Thread one = new Thread(r);
		Thread two = new Thread(r);
		one.setName("Fred");
		two.setName("Lucy");
		one.start();
		two.start();
	}

	public void run() {
		for (int x = 0; x < 5; x++) {
		makeWithdrawal(10);
			if (acct.getBalance() < 0) {
				System.out.println("account is overdrawn!");
			}
		}
	}

	private void makeWithdrawal(int amt) {
		if (acct.getBalance() >= amt) {
			System.out.println(Thread.currentThread().getName()
			+ " is going to withdraw");
				try {
				Thread.sleep(500);
				} catch(InterruptedException ex) { }
			acct.withdraw(amt);
			System.out.println(Thread.currentThread().getName()
			+ " completes the withdrawal");
		} else {
			System.out.println("Not enough in account for "
			+ Thread.currentThread().getName()
			+ " to withdraw " + acct.getBalance());
		}
	}
}
```

Se obtendría la siguiente salida:

```
Fred is going to withdraw
Lucy is going to withdraw
Fred completes the withdrawal
Lucy completes the withdrawal
Fred is going to withdraw
Lucy is going to withdraw
Lucy completes the withdrawal
Lucy is going to withdraw
Fred completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Lucy completes the withdrawal
account is overdrawn!
account is overdrawn!
Not enough in account for Fred to withdraw -10
Not enough in account for Lucy to withdraw -10
account is overdrawn!
account is overdrawn!
Not enough in account for Fred to withdraw -10
Not enough in account for Lucy to withdraw -10
account is overdrawn!
account is overdrawn!
```

Como Fred y Lucy están accesando a la cuenta al mismo tiempo, están compartiendo el dato del balance y no hay control para que se tomen turnos para realizar la operación y se podría decir que la están intentando hacer al mismo tiempo, por lo tanto no se dan cuenta de las actualizaciones que se hacen en la cuenta.

La solución a éste problema es simple. Se debe garantizar que no se hagan las operaciones al mismo tiempo sobre la cuenta como si fueran dos cosas independientes que no se afectan entre sí, a esto se le llama una "operación atómica", ya que se supone que un átomo es indivisible. Con esto se va a garantizar que Fred no va a poder actuar sobre la cuenta hasta que Lucy complete sus operaciones y viceversa.

Para arreglar los problemas de Fred y Lucy hay que hacer todas las variables privadas y sincronizar todos los métodos que modifiquen dichas variables.

```
private synchronized void makeWithdrawal(int amt) {
      if (acct.getBalance() >= amt) {
            System.out.println(Thread.currentThread().getName() +
            " is going to withdraw");
            try {
                  Thread.sleep(500);
            } catch(InterruptedException ex) { }
            acct.withdraw(amt);
            System.out.println(Thread.currentThread().getName() +
            " completes the withdrawal");
      } else {
            System.out.println("Not enough in account for "
            + Thread.currentThread().getName()
            + " to withdraw " + acct.getBalance());
      }
}
```

Con esto se garantiza que sólo un thread (Fred o Lucy) empiecen la operación de retiro, y que sólo se puede utilizar el método hasta que el otro lo termine de utilizar.

Del código anterior se tiene la siguiente salida:

```
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
```

3.7.1 Sincronización y Locks.

Todos los objetos en Java tienen una bandera asociada llamada lock, La palabra syncronized habilita el uso de esta bandera y proporciona un acceso exclusivo para el código donde se comparten datos.

Cuando un thread busca la palabra syncronized trat de obtener la bandera antes de continuar la ejecución, si la bandera no está presente, no puede continuar su ejecución. Una clase puede tener tanto métodos sincronizados, como no sincronizados. Se pueden sincronizar tanto métodos como pequeños fragmentos de código.

```
class SyncTest {
	public void doStuff() {
		System.out.println("not synchronized");
		synchronized(this) {
			System.out.println("synchronized");
		}
	}
}
```

También los métodos estáticos pueden ser sincronizados. Sólo hay una copia de los datos estáticos que se deben proteger. Por lo tanto sólo se necesita una sincronización a toda la clase. Toda clase cargada en Java tiene su correspondiente instancia de java.lang.Class representando esa clase. Esta instancia es la que se debe proteger mediante la sincronización. De igual forma se puede hacer de dos maneras:

Sincronizando el método:

```
public static synchronized int getCount() {
	return count;
}
```

O

Sincronizando parte del código.

```
public static int getCount() {
```

```
        synchronized(MyClass.class) {
                return count;
        }
}
```

De esta manera se le está diciendo a la JVM ve y encuentra la instancia de Class que representa MyClass. Lo anterior también se puede hacer de la siguiente forma:

```
public static void classMethod() {
        Class cl = Class.forName("MyClass");
        synchronized (cl) {
                // do stuff
        }
}
```

3.7.2 Thread Deadlock.

Esto sucede cuando un thread esta esperado por la bandera, pero el otro thread también está esperando por la misma bandera, de tal manera que los dos threads están bloqueados. Detectar éste tipo de problemas es asunto del programador, que debe asegurarse que el deadlock no se presente. Si se tienen múltiples objetos que se necesitan ser sincronizados, hay que tomar una desición general sobre el orden en que e obtendrán las banderas.

3.7.3 Interacción con los Threads.

La último cosa que se tiene que sabes sobre los threads, es la manera en que interactúan entre sí. Para esto se tiene los métodos wait(), notify() y notifyAll() de la clase Object. Por ejemplo, si un proceso se encarga de envíar correos y otro de procsarlos, el procesador tiene que estar revisando constantemente hasta que existan correos que procesar. Usando wait() y notify(), el procesador le dice al enviador, "No tengo por qué estar perdiendo el tiempo chocando cada dos segundos si hay un mail, por favor notificame cuando envíes uno", de esta manera el thread procesador puede regresa a su estado runnable.

Hay que tener presente que tanto wait(), notify() y notifyall() deben ser llamados en un contexto sincronizado.

El siguiente programa contiene dos objetos con threads. ThreadA contiene el main y ThreadB tiene un thread que se encarga de calcular la suma de todos los número del 0 al 99. Tan pronto como se llama a start(), el ThredA continúa con la siguiente línea de código de su propia clase.

Hay que notar que en ThreadA se sincroniza el objeto b, esto es porque ThreadA debe obtener la bandera de B. Cuando se llama a wait() o notify(), el thread tiene que ser el dueño de la bandera del otro objeto.

```
class ThreadA {
     public static void main(String [] args) {
```

```
            ThreadB b = new ThreadB();
            b.start();
            synchronized(b) {
            try {
                  System.out.println("Waiting for b to complete...");
                  b.wait();
            } catch (InterruptedException e) {}
            System.out.println("Total is: " + b.total);
      }
}
}

class ThreadB extends Thread {
      int total;
            public void run() {
             synchronized(this) {
                  for(int i=0;i<100;i++) {
                        total += i;
                  }
                  notify();
             }
      }
}
```

3.7.4 Usando notifyAll().

En otros escenarios es necesario notificar a todos los threads que están esperando por un objeto en particular. En éste caso se utiliza notifyAll, con esto todos los objetos vuelven a un estado runnable. Cuando todos los threads son notificados, estos compiten por obtener la bandera.

```
class Reader extends Thread {
      Calculator c;
      public Reader(Calculator calc) {
            c = calc;
      }
      public void run() {
            synchronized(c) {
            try {
                  System.out.println("Waiting for calculation...");
                  c.wait();
            } catch (InterruptedException e) {}
                  System.out.println("Total is: " + c.total);
            }
      }

      public static void main(String [] args) {
            Calculator calculator = new Calculator();
            new Reader(calculator).start();
            new Reader(calculator).start();
            new Reader(calculator).start();
            calculator.start();
      }
}
class Calculator extends Thread {
      int total;

      public void run() {
            synchronized(this) {
                  for(int i=0;i<100;i++) {
                        total += i;
                  }
                  notifyAll();
            }
      }
}
```

5. Creación de GUIs

El objetivo original del diseño de la librería de la interfaz gráfica de usuario (GUI) de Java 1.0 era permitir al programador construir una aplicación gráfica que se viera bien en todas las plataformas. Ese objetivo no fue cumplido, de hecho, la herramienta que fue diseñada para cumplir con las expectativas, Abstract Window Toolkit (AWT) , resultó decepcionante debido a que las aplicaciones resultantes se veían bastante mediocres en cualquier plataforma además de ser una herramienta bastante restrictiva: sólo permitía el uso de cuatro fuentes y no permitía el acceso a cualquier elemento más sofisticado que nos brindara el Sistema Operativo.

Su modelo tampoco era orientado a objetos. La situación mejoró de manera significativa con la salida de Java 1.1. El modelo de eventos de AWT se volvió mucho más claro además de la incorporación de los Java Beans, que son componentes orientadas a la creación de ambientes visuales. Pero fue finalmente Java 2 quien terminó el trabajo esencialmente reemplazando al viejo AWT con las llamadas Java Fundation Classes (JFC), de las cuales forma parte la GUI conocida como Swing.

Swing cumple con todas las expectativas planteadas originalmente y además va más allá ya que sin lugar a dudas es una herramienta de programación que nos permite crear GUI modernas y competitivas. La librería de Swing contiene desde simples botones hasta imágenes, árboles y tablas.

Creación de una interface gráfica de usuario.

Para construir una interface gráfica de usuario hace falta:

1. Un "contenedor" o ***container***, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos.
2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc.
3. El ***modelo de eventos***. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el ***evento*** correspondiente, que el sistema operativo transmite al AWT. El AWT crea un ***objeto*** de una determinada clase de evento, derivada de ***AWTEvent***. Este ***evento*** es transmitido a un determinado ***método*** para que lo gestione.

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos (***event sources***) "registran" los objetos que habrán de gestionarlos (***event listeners***), para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir.

1. Introducción

Java es un lenguaje con el objetivo de ser multiplataforma, el diseño de su toolkit para programación de GUIs se hizo pensando en que las aplicaciones tuvieran un buen aspecto en cualquier plataforma pero independiente de cualquier GUI específico. Este toolkit se denominó AWT 1.0 (Abstract Window Toolkit):

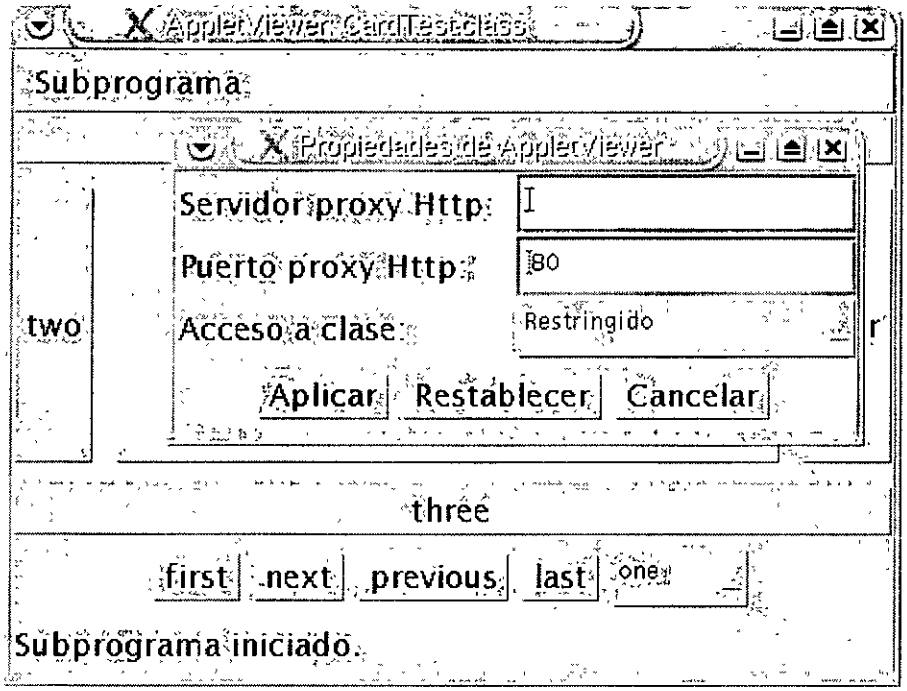


Fig.1: AWT 1.0

Realmente las aplicaciones AWT 1.0 tenían un aspecto rudimentario y un escaso número de elementos. Además su diseño interno era muy deficiente.

La situación mejoró algo con AWT 1.1, pero no fue hasta Java 1.2 cuando apareció Swing (Realmente Swing no sustituye a AWT. Swing está construido como una capa sobre AWT, que sigue estando disponible en la biblioteca de clases Java), un toolkit completamente nuevo, con un diseño interno orientado a componentes y un look mucho más satisfactorio:

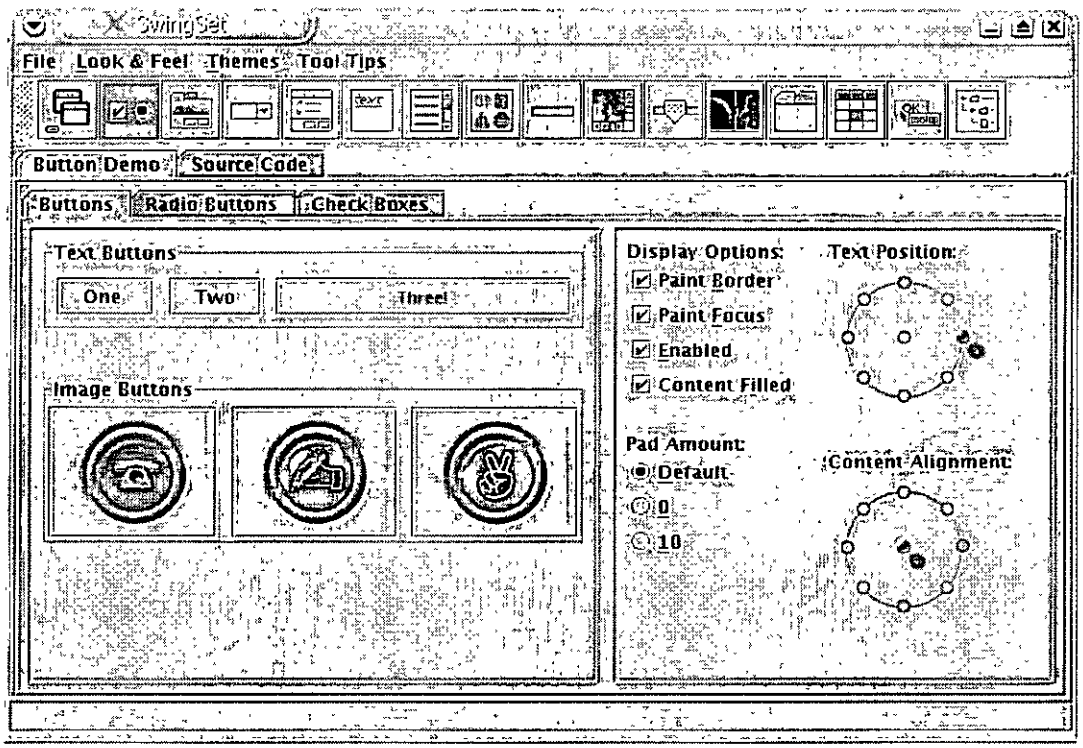


Fig.2: Swing

A pesar de que Swing tiene un estilo visual propio por defecto, puede también utilizar un aspecto "Motif", "Windows" o "Apple"; estos últimos sólo en las plataformas correspondientes. Además puede cambiar de aspecto en tiempo de ejecución.

Aspecto Swing por defecto:

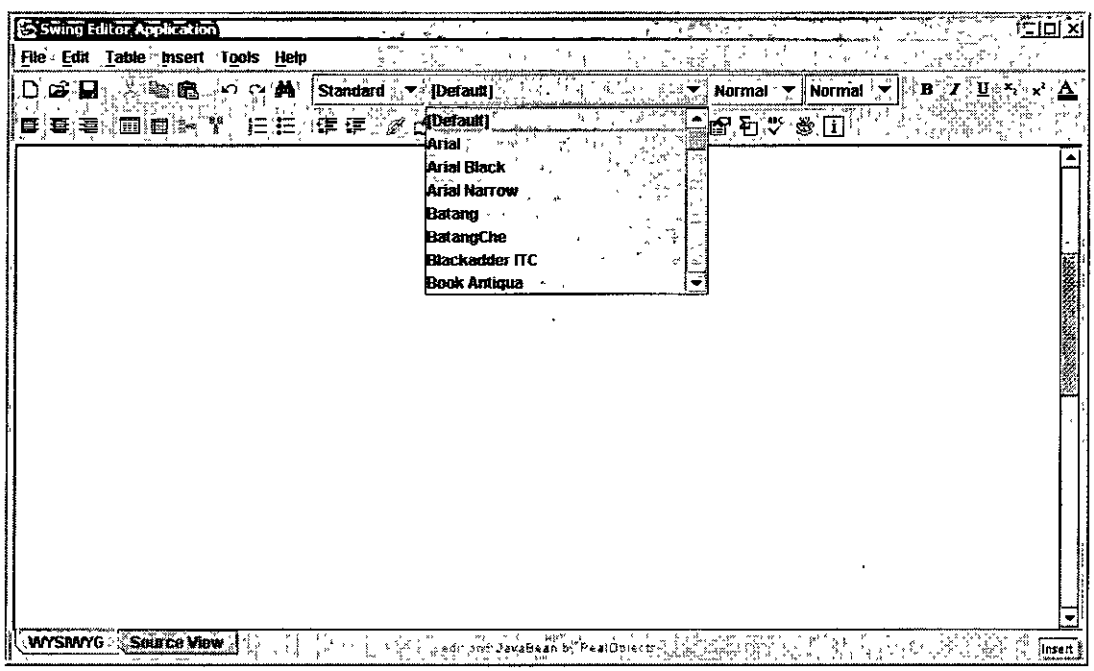


Fig.3: Swing por defecto

Aspecto Motif:

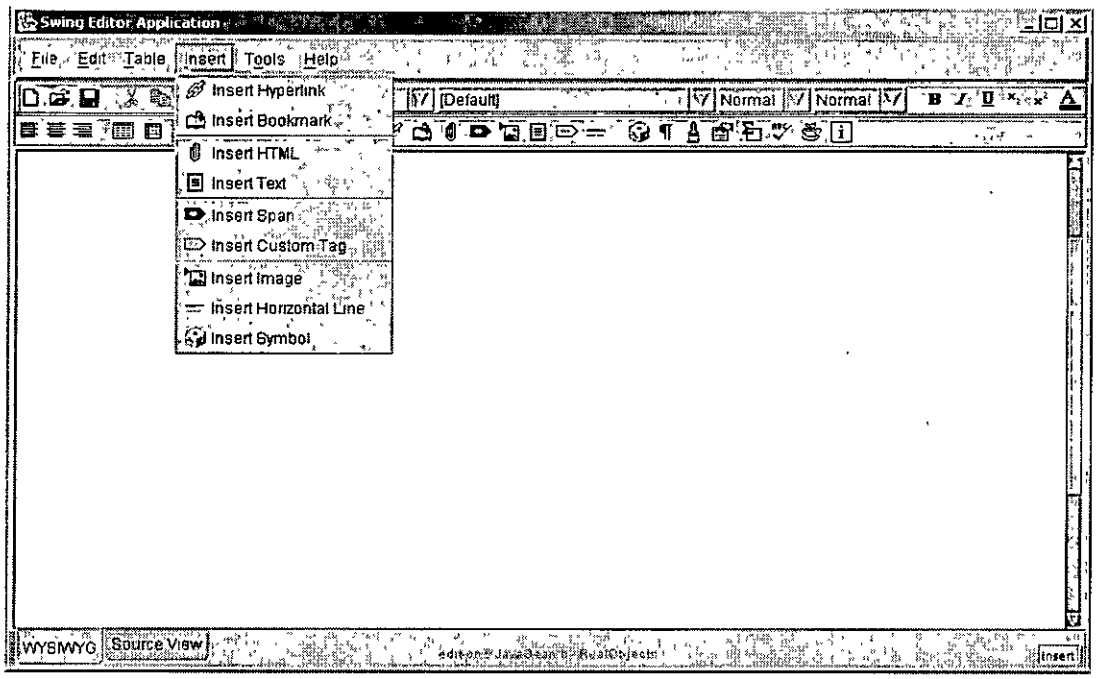


Fig.4: Swing Motif

Aspecto Windows:

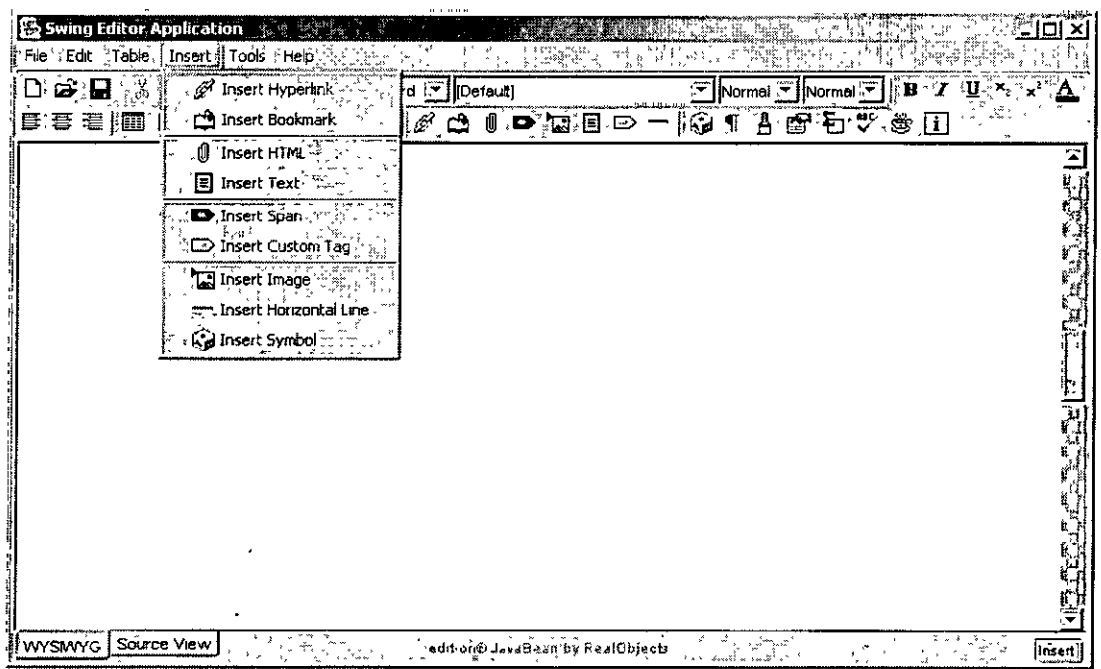


Fig.5: Swing Windows

2. AWT (Abstract Window Toolkit)

Las JFC (Java Foundation Classes) son parte de la API de Java, compuesto por clases que sirven para crear interfaces gráficas visuales para las aplicaciones y applets de Java. JFC fue presentado por primera vez en la conferencia de desarrolladores JavaOneSM de 1997. Tanto AWT como Swing, son paquetes gráficos contenidos en las JFC.

AWT es por tanto un conjunto de herramientas GUI diseñadas para trabajar con múltiples plataformas. Este paquete viene incluido en la API de Java como java.awt ya desde su primera versión, con lo que las interfaces generadas con esta biblioteca funcionan en todos los entornos Java disponibles. También funcionan en navegadores que soporten Java lo que les hace especialmente eficientes para la creación de applets. En la siguiente figura podemos observar las principales clases de AWT:

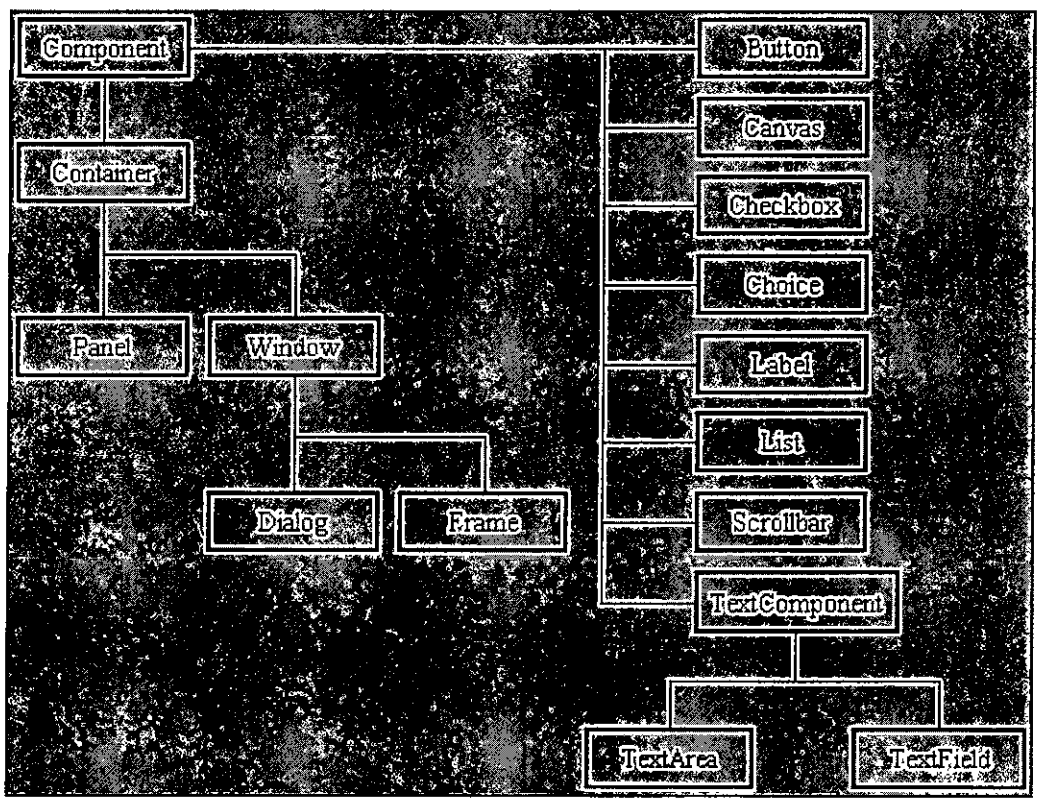


Fig.6: Principales clases AWT

2.3. Swing

El paquete Swing es el nuevo paquete grafico que ha aparecido en la versión 1.2 de Java. Esta compuesto por un amplio conjunto de componentes de interfaces de usuario y que pretenden funcionar en el mayor número posible de plataformas. Cada uno de los

componentes de este paquete puede presentar diversos aspectos y comportamientos en función de una biblioteca de clases.

En la versión 1.0 de Swing, que corresponde a la distribuida en la versión 1.2 del API de Java se incluyen tres bibliotecas de aspecto y comportamiento para Swing:

metal.jar: Aspecto y comportamiento independiente de la plataforma.

motif.jar: Basado en la interfaz Sun Motif.

windows.jar: Muy similar a las interfaces Microsoft Windows 95.

2.3.1 Diferencias entre Swing y AWT

Los componentes Swing, su escritura, su manipulación y se despliegue son completamente en Java (ofrecen mayor portabilidad y flexibilidad). Por ello se les llama componentes puros de Java. Como están completamente escritos en Java y no les afectan las complejas herramientas GUI de la plataforma en la que se utilizan, también se les conoce comúnmente como componentes ligeros. De hecho de las principales diferencias entre los componentes de *java.awt* y de *javax.swing* es que los primeros están enlazados directamente a las herramientas de la interfaz gráfica de usuario de la plataforma local.

Por lo tanto, un programa en Java que se ejecuta en distintas plataformas Java tiene una apariencia distinta e incluso, algunas veces hasta la iteraciones del usuario son distintas en cada plataforma (a la apariencia y a la forma en que el usuario interactúa con el programa se les conoce como la "apariencia visual del programa").

Sin embargo los componentes Swing permiten al programador especificar una apariencia visual distinta para cada plataforma, una apariencia visual uniforme entre todas las plataformas, o incluso puede cambiar la apariencia visual mientras el programa se ejecuta.

A los componentes AWT que se enlazan a la plataforma local se les conoce como componentes pesados (dependen del sistema de ventanas de la plataforma local para determinar su funcionalidad y su apariencia visual). Cada componente pesado tiene un componente asociado (del paquete *java.awt.peer*), el cual es responsable de las interacciones entre el componente pesado y la plataforma local para mostrarlo y manipularlo.

Varios componentes Swing siguen siendo pesados. En particular las subclases de *java.awt.Window*, que muestran ventanas en la pantalla, aun requieren de una interacción directa con el sistema de ventanas local. (En consecuencia, los componentes pesados de Swing son menos flexibles)

Otras ventajas de Swing respecto a AWT son:

- Amplia variedad de componentes: En general las clases que comiencen por "J" son componentes Swing que se pueden añadir a la aplicación. Por ejemplo: JButton.

- Aspecto modificable (look and feel): Se puede personalizar el aspecto de las interfaces o utilizar varios aspectos que existen por defecto (Metal Max, Basic Motif, Window Win32).

- Arquitectura Modelo-Vista-Controlador: Esta arquitectura da lugar a todo un enfoque de desarrollo muy arraigado en los entornos gráficos de usuario realizados con técnicas orientadas a objetos. Cada componente tiene asociado una clase de modelo de datos y una interfaz que utiliza. Se puede crear un modelo de datos personalizado para cada componente, con sólo heredar de la clase modelo.

- Gestión mejorada de la entrada del usuario: Se pueden gestionar combinaciones de teclas en un objeto KeyStroke y registrarlo como componente. El evento se activará cuando se pulse dicha combinación si está siendo utilizado el componente, la ventana en que se encuentra, algún hijo del componente.

- Objetos de acción (action objects): Estos objetos cuando están activados (enabled) controlan las acciones de varios objetos componentes de la interfaz. Son hijos de ActionListener.

- Contenedores anidados: Cualquier componente puede estar anidado en otro. Por ejemplo, un gráfico se puede anidar en una lista.

- Escritorios virtuales: Se pueden crear escritorios virtuales o "interfaz de múltiples documentos" mediante las clases JDesktopPane y JInternalFrame.

- Bordes complejos: Los componentes pueden presentar nuevos tipos de bordes. Además el usuario puede crear tipos de bordes personalizados.

- Diálogos personalizados: Se pueden crear multitud de formas de mensajes y opciones de diálogo con el usuario, mediante la clase JOptionPane.

- Clases para diálogos habituales: Se puede utilizar JFileChooser para elegir un archivo, y JColorChooser para elegir un color.

- Componentes para tablas y árboles de datos: Mediante las clases JTable y JTree.

- Potentes manipuladores de texto: Además de campos y áreas de texto, se presentan campos de sintaxis oculta JPassword, y texto con múltiples fuentes JTextPane. Además hay paquetes para utilizar archivos en formato HTML o RTF.-Capacidad para "deshacer": En gran variedad de situaciones se pueden deshacer las modificaciones que se realizaron.

2.3.2 Herencia

La siguiente relación muestra una jerarquía de herencia de las clases que definen los atributos y comportamientos comunes para la mayoría de los componentes Swing. Cada clase aparece con su nombre y con el nombre completo de su paquete. La mayor parte de la funcionalidad de cada componente GUI se deriva de esas clases.

java.lang.Object -> java.awt.Component -> java.awt.Container ->javax.swing.JComponent

Los componentes Swing que corresponden a subclases de JComponent tienen muchas características, las cuales incluyen:

- Una apariencia visual adaptable que puede utilizarse para personalizar la apariencia visual cuando el programa se ejecuta en distintas plataformas
- Teclas de acceso directo (llamadas mnemónicos) para acceder directamente a los componentes GUI a través del teclado.
- Herramientas para manejo de eventos comunes, para casos en los que componentes GUI inician las mismas acciones en un programa.
- Breves descripciones del propósito de un componente GUI (que se conocen como cuadros de información de herramientas) que aparecen cuando el cursor del ratón se posiciona sobre el componente durante un periodo de tiempo corto.
- Soporte para tecnologías de asistencia tales como lectores de pantalla braille para personas ciegas.
- Soporte para localización de la interfaz de usuario; es decir, para personalizar la interfaz de usuario de manera que aparezca en distintos lenguajes y convenciones culturales.

2.3.3 Componentes de Swing

Contenedores básicos:

-JFrame: Representa una ventana básica, capaz de contener otros componentes. Casi todas las aplicaciones construyen al menos un Jframe.

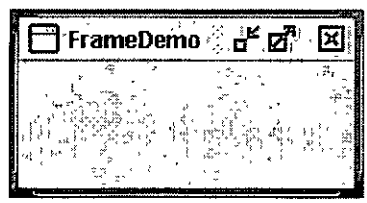


Fig.7: JFrame

-JDialog, JOptionPane, etc: Los cuadros de diálogo son JFrame restringidos, dependientes de un JFrame principal. Los JOptionPane son cuadros de diálogo sencillos predefinidos para pedir confirmación, realizar advertencias o notificar errores. Los JDialog son cuadros de diálogo generales, normalmente utilizados para peticiones de datos.

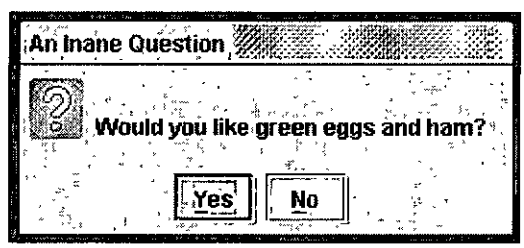


Fig.8: JDialog

-JInternalFrame: Consiste simplemente en una ventana hija, que no puede salir de los límites marcados por la ventana principal. Es muy común en aplicaciones que permiten tener varios documentos abiertos simultáneamente.

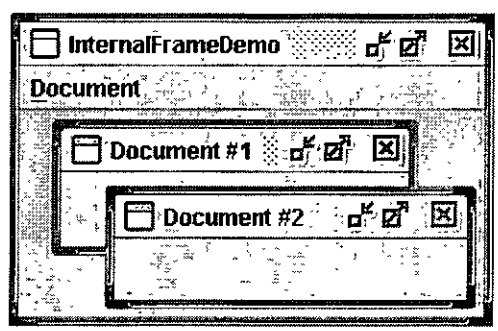


Fig.9: JInternalFrame

-JPanel: Un panel sirve para agrupar y organizar otros componentes. Puede estar decorado mediante un borde y una etiqueta.

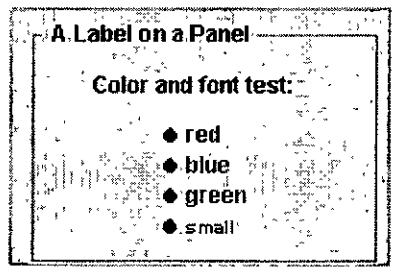


Fig.10: JPanel

-JScrollPane: Es un panel que permite visualizar un componente de un tamaño mayor que el disponible, mediante el uso de barras de desplazamiento.

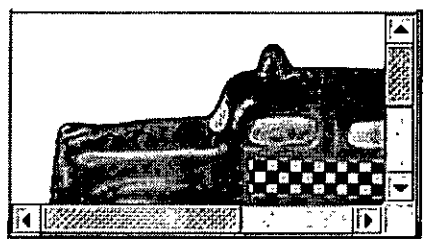


Fig.11: JScrollPane

-JSplitPane: Permite visualizar dos componentes, uno a cada lado, con la posibilidad de modificar la cantidad de espacio otorgado a cada uno.

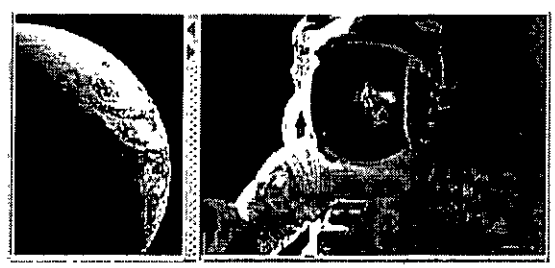


Fig.12: JSplitPane

-JTabbedPane: Permite definir varias hojas con pestañas, que pueden contener otros componentes. El usuario puede seleccionar la hoja que desea ver mediante las pestañas.

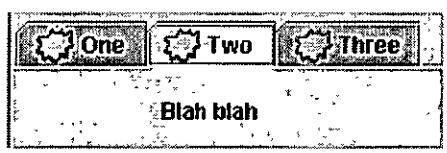


Fig.13: JTabbedPane 1

-JToolBar: Es un contenedor que permite agrupar otros componentes, normalmente botones con iconos en una fila o columna. Las barras de herramientas tienen la particularidad de que el usuario puede situarlas en distintas configuraciones sobre el frame principal.

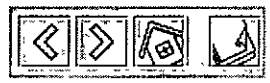


Fig.14: JToolBar 1

Controles básicos:

-JButton, JCheckBox, JRadioButton: Distintos tipos de botones. Un check box sirve para marcar una opción. Un radio button permite seleccionar una opción entre varias disponibles.

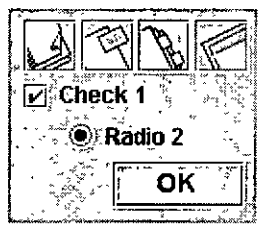


Fig.15: JButton, JCheckBox, JRadioButton

-JComboBox: Las combo boxes o listas desplegables que permiten seleccionar un opción entre varias posibles.

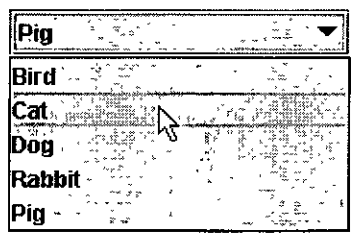


Fig.16: JComboBox

-JList: Listas que permiten seleccionar uno o más elementos.

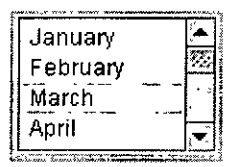


Fig.17: JList

-JTextField, JFormattedTextField, JpasswordField: Distintos tipos de editores. JFormattedTextField permite indicar el conjunto de caracteres legales que pueden introducirse. JPasswordField no muestra el contenido.

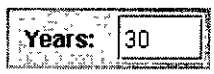


Fig.18: JTextField

-JSlider: Un slider permiten introducir un valor numérico entre un máximo y un mínimo de manera rápida.

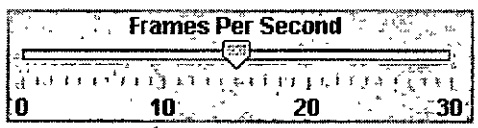


Fig.19: JSlider

-JSpinner: Permiten seleccionar un valor entre un rango de opciones posibles, al igual que las listas desplegables, aunque no muestran tal lista. Los valores cambian al pulsar los botones de desplazamiento. También se puede introducir un valor directamente.

Fig.20: JSpinner

-Menús desplegables. Existen dos tipos de menús:

JMenuBar, que consiste en una barra de menús desplegables en la parte superior de la aplicación, y JPopupMenu, un menú que se obtiene al pulsar con el botón derecho del ratón sobre una zona determinada. Los menús están compuestos por distintos ítems:

JSeparator (una línea de separación entre opciones), JMenuItem (una opción ordinaria), JMenu (un submenu), JCheckboxMenuItem (un opción en forma de check box) o finalmente JRadioButtonMenuItem (una opción en forma de radio button).

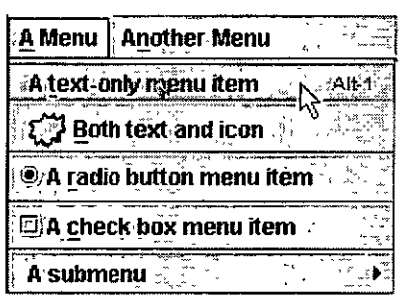


Fig.21: Menus

Controles especializados:

-JColorChooser: Consiste en un selector de colores.

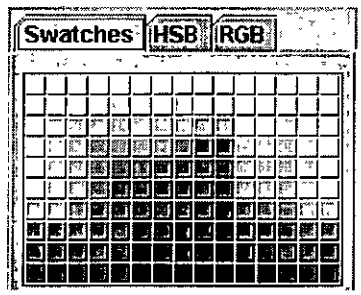


Fig.22: JColorChooser

-JFileChooser: Permite abrir un cuadro de diálogo para pedir un nombre de archivo.

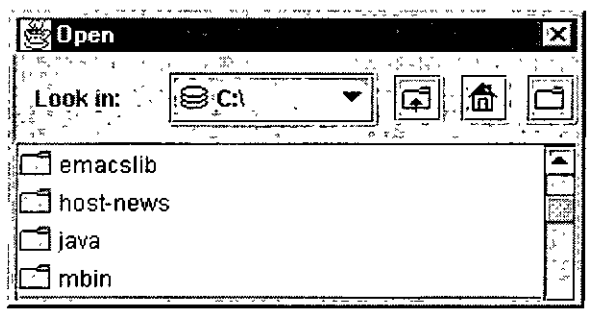


Fig.23: JFileChooser

-JTree: Su función es mostrar información de tipo jerárquico.

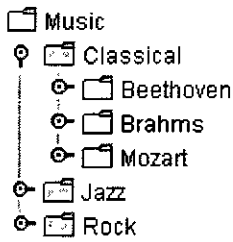


Fig.24: JTree

Controles básicos no interactivos:

-JLabel: Permite situar un texto, un texto con una imagen o una imagen únicamente en la ventana. No son interactivos y puede utilizarse código HTML para escribir texto en varias líneas y con varios atributos.

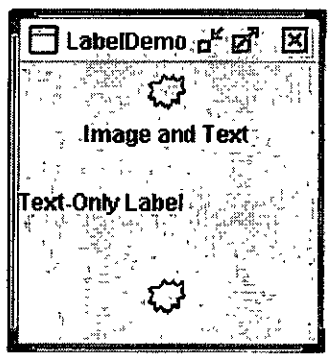


Fig.25: JLabel

-JProgressBar: Permite mostrar que porcentaje del total de una tarea a realizar ha sido completado.

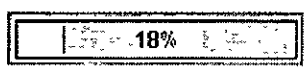


Fig.26: JProgressBar

-JToolTip: Consiste en una etiqueta de ayuda que surge al cabo de uno segundos sobre la posición apuntada por el cursor. Normalmente no es necesario utilizar directamente la clase JToolTip, se puede establecer para cualquier componente de la ventana mediante:

e.setToolTipText ("Esta es la etiqueta");

Fig.27: JToolTip

2.4 Introducción

Ahora vamos a profundizar en uno de los controles básicos de Swing: los botones. Los botones son una de las piezas claves en la mayoría de interfaces gráficas de usuario ya que permiten al usuario, por ejemplo, tanto elegir entre varias opciones (radio buttons), seleccionar las características que quiere que tenga su ventana da aplicación (check boxes), o simplemente trasladar al software su deseo de realizar una determinada acción.

Por eso además de saber de donde provienen las clases JRadioButton, JCheckBox y JButton, es necesario que hagamos un breve estudio sobre la gestión de eventos, para comprender que además de situar y crear correctamente nuestro botón, hay que indicarle lo que queremos que haga cuando se realiza una acción determinada sobre él.

3.1. Herencia

La siguiente relación nos muestra la herencia de las clases que estudiaremos posteriormente:

javax.swing.JCheckbox y javax.swing.JRadioButton son subclases de javax.swing.JToggleButton

javax.swing.JToggleButton y javax.swing.JButton son subclases de javax.swing.AbstractButton

javax.swing.AbstractButton es subclase de javax.swin.JComponent

3.2. Eventos

Las GUI's están controladas por eventos (generan eventos cuando el usuario interactúa con la GUI). Siempre que ocurre una interacción con el usuario se envía un evento al programa. La información de los eventos de la GUI se almacena en un objeto de una clase que extiende a AWTEvent.

Los eventos que vamos a gestionar en los ejemplos correspondientes a nuestros componentes pertenecen al paquete java.awt.event. También se han agregado tipos de eventos adicionales, específicos para varios tipos de componentes Swing. Estos eventos se definen en el paquete *javax.swing.event*.

Para procesar un evento de interfaz gráfica de usuario, el programador debe realizar dos tareas clave:

-Registrar un componente que escuche eventos (es un objeto de una clase que implementa una o más de las interfaces que escuchan eventos correspondientes a los paquetes *java.awt.event* y *javax.swing.event*).

-Implementar un manejador de eventos (método que se invoca automáticamente en respuesta a un tipo específico de evento).

Cualquier clase que implemente a una interfaz deberá definir todos los métodos de esa interfaz; en caso contrario, será una clase abstract y no podría utilizarse para crear objetos.

Al uso de componentes que escuchan eventos en el manejo de eventos se conoce como "modelo de delegación de eventos"; el procesamiento de un evento se delega a un objeto específico en el programa.

Cuando ocurre un evento, el componente GUI que interactuó con el usuario, notifica a sus componentes de escucha registrados, por medio de una llamada al método manejador de eventos apropiado de cada componente de escucha.

Existen ciertos "event listeners" que son comunes a todos los componentes. Los que vienen a continuación (se usan en lso ejemplos) son producidos específicamente por los componentes JRadioButton, JCheckBox y JButton:

-ActionListener: captura cierto tipo de acción realizada sobre ciertos componentes. Por ejemplo, pulsar un botón, seleccionar un elemento en una lista desplegable o una opción en un menú.

-ItemListener: recoge el cambio de estado en un componente tipo on/off: check boxes, radio buttons y listas desplegables.

Si la interfaz captura un único tipo de evento, como ActionListener, normalmente tendrá una única operación a implementar:

void actionPerformed (ActionEvent e)

Existen dos formas básicas de implementación de nuestro event listener:

-Utilizar la clase principal frame, frame interior o cuadro de diálogo como receptor de los eventos de los subcomponentes. Para ello basta con hacer que implemente la correspondiente interfaz.

Esta opción es simple y eficiente, sin embargo, puede darse el caso de que las operaciones implementadas de la interfaz reciban eventos de varios componentes. Además la clase no va a poder redefinir un adaptador (para las interfaces con más de una operación como WindowListener, la librería proporciona una clase adaptadora (WindowAdapter) que implementa la interfaz mediante implementaciones vacías por defecto, de forma que podamos heredar y redefinir la operación que nos interesa en lugar de tener que implementar toda la interfaz), al heredar ya de una clase previamente.

-Construir una pequeña clase interior que implemente la interfaz. Esta opción es más flexible y permite definir directamente un procesamiento único para cada evento/componente que lo genera. Sin embargo la definición de nuevas clases y la creación de objetos interiores la hacen menos eficiente (Es la opción que utilizaremos en los ejemplos).

Una vez implementado el listener, indicaremos al componente que lo utilice para notificar sus eventos mediante:

componente.addxxxxListener (objetoListener)

donde xxxx será el identificador del tipo de listener.

Un componente puede ser conectado a varios listeners del mismo o distintos eventos. De la misma forma, un listener puede ser conectado a varios componentes al mismo tiempo. En estos casos para determinar dentro del listener cuál es el componente que ha enviado el evento podemos utilizar la operación getSource() del evento recibido. Esta situación es muy frecuente cuando la clase principal hace de listener de los subcomponentes.

3.4. JButton

Un botón es un componente en el que el usuario hace clic para desencadenar una acción especifica (genera un evento ActionEvent).

Al texto en la cara de un objeto JButton se le llama etiqueta del botón. Tener mas de un objeto JButton con la misma etiqueta hace que los objetos JButton sean ambiguos para el usuario (cada etiqueta de botón debe ser única).

Un objeto JButton puede mostrar objetos Icon, esto proporciona un nivel adicional de interactividad visual. También puede tener un objeto de sustitución, que es un objeto Icon que aparece cuando el ratón se posiciona sobre el botón; el icono en el botón cambia a medida que el ratón se aleja y se acerca al área del botón en la pantalla.

3.4.1 Métodos y constructores

Ahora que sabemos como es visualmente cada uno de los componentes (visto en el punto 2.3.3), hay que destacar los métodos y constructores más relevantes que nos permitirán configurar los componentes según nuestras necesidades.

-Contenido del botón

Método o Constructor	Propósito
JButton(String,Icon) JButton(String) JButton(Icon) JButton()	Crea un ejemplar de JButton, lo inicializa para tener el texto/imagen especificado.
void setText(String) String getText()	Selecciona u obtiene el texto mostrado en el botón.
void setIcon(Icon) Icon getIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando no está seleccionado o pulsado.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando está desactivado. Si no se especifica una imagen, el aspecto y comportamiento crea una por defecto.
void setPressedIcon(Icon) Icon getPressedIcon()	Selección u obtiene la imagen mostrada por el botón cuando está pulsado.
void setSelectedIcon(Icon) Icon getSelectedIcon() void setDisabledSelectedIcon(Icon) Icon getDisabledSelectedIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando está seleccionado. Si no se especifica una imagen de botón desactivado seleccionado, el aspecto y comportamiento crea una manipulando la imagen de seleccionado.
setRolloverEnabled(boolean) boolean getRolloverEnabled() void setRolloverIcon(Icon) Icon getRolloverIcon() void setRolloverSelectedIcon(Icon) Icon getRolloverSelectedIcon()	Utiliza setRolloverEnabled(true) y setRolloverIcon(someIcon) para hacer que el botón muestre el icono especificado cuando el cursor pasa sobre él.

-Ajustes sobre la apariencia del botón

Método o constructor	Propósito
void setHorizontalAlignment(int) void setVerticalAlignment(int) int getHorizontalAlignment() int getVerticalAlignment()	Selecciona u obtiene dónde debe situarse el contenido del botón. La clase AbstractButton permite uno de los siguientes valores para alineamineto horizontal: LEFT, CENTER (por defecto), y LEFT. Para alineamiento vertical: TOP, CENTER (por defecto),
void setHorizontalTextPosition(int) void setVerticalTextPosition(int) int getHorizontalTextPosition() int getVerticalTextPosition()	Selecciona u obtiene dónde debería situarse el texto del botón con respecto a la imagen. La clase AbstractButton permite uno de los siguientes valores para alineamineto horizontal: LEFT, CENTER (por defecto), y LEFT. Para alineamiento vertical: TOP, CENTER (por defecto), y BOTTOM.
void setMargin(Insets) Insets getMargin()	Selecciona u obtiene el número de píxeles entre el borde del botón y sus contenidos.
void setFocusPainted(boolean) boolean isFocusPainted()	Selecciona u obtiene si el botón debería parecer diferente si obtiene el foco.
void setBorderPainted(boolean) boolean isBorderPainted()	Selecciona u obtiene si el borde del botón debería dibujarse.

-Funcionalidad del botón

Método o Constructor	Propósito
void setMnemonic(char) char getMnemonic()	Selecciona la tecla alternativa para pulsar el botón.
void setActionCommand(String) String getActionCommand(void)	Selecciona u obtiene el nombre de la acción realizada por el botón.
void addActionListener(ActionListener) ActionListener removeActionListener()	Añade o elimina un objeto que escucha eventos action disparados por el botón.
void addItemListener(ItemListener) ItemListener removeItemListener()	Añade o elimina un objeto que escucha eventos items disparados por el botón.
void setSelected(boolean) boolean isSelected()	Selecciona u obtiene si el botón está seleccionado. Tiene sentido sólo en botones que tienen un estado on/off, como los checkbox.
void doClick() void doClick(int)	Realiza un "click". El argumento opcional especifica el tiempo (en milisegundos) que el botón debería estar pulsado.

3.4.2 Ejemplo

En este ejemplo igual que en los demás se va a crear una pequeña aplicación que nos muestre una iniciación a como crear el correspondiente componente.

Vamos a crear dos objetos JButton y demostramos que soportan el despliegue de objetos Icon. El manejo de eventos se lleva a cabo mediante una sola instancia de la clase interna ManejadorBoton.

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

public class PruebaBoton extends JFrame {

private JButton botonSimple, botonElegante;

public PruebaBoton(){

super( "Prueba de botones" );
Container c = getContentPane();
c.setLayout( new FlowLayout() );
```

```
// crea los botones
botonSimple = new JButton( "Boton simple" );

//agregamos el botón al panel de contenido
c.add( botonSimple );


/* creamos dos objetos ImageIcon que representan al objeto Icon predeterminado
y al objeto Icon de sustitucion para el botonElegante. Los gif se presupone que
están en el mismo
directorio que la aplicación que las utiliza  */

Icon icono1 = new ImageIcon( "icono1.gif" );
Icon icono2 = new ImageIcon( "icono2.gif" );

/* crea el boton con el gif y el texto (texto a la derecha del icono como
predeterminado) */

botonElegante = new JButton("Boton elegante",icono1);

/* se utiliza el método setRollOverIcon heredado de AbstractButton para
especificar la imagen que aparece cuando el ratón se posiciona sobre el botón
*/

botonElegante.setRolloverIcon( icono2 );
// agregamos el boton al panel de contenido
c.add( botonElegante );


/* crea una instancia de la clase interna ManejadorBoton para usarla en el
manejo de eventos de botón */



ManejadorBoton manejador = new ManejadorBoton();
botonElegante.addActionListener( manejador ); botonSimple.addActionListener(
manejador );
setSize( 300, 100 );
show();
} // fin del constructor de PruebaBoton

public static void main( String args[] )
{

      PruebaBoton ap = new PruebaBoton();
      ap.addWindowListener(
            new WindowAdapter() {
                  public void windowClosing( WindowEvent e )
                  {
                        System.exit( 0 );
                  } // fin del método windowClosing
            } // fin de la clase interna anónima
      ); // fin de addWindowListener
} // fin de main
```

```
// clase interna para el manejo de eventos de botón


 private class ManejadorBoton implements ActionListener {

      public void actionPerformed( ActionEvent e )
      {
      /* mostramos un cuadro de dialogo de mensaje que contiene la etiqueta del
      boton que se pulso */

      JOptionPane.showMessageDialog( null, "Usted oprimio:" +
      e.getActionCommand() );

      }

   }

}
```

3.5. JCheckBox

La versión Swing soporta botones checkbox con la clase JCheckBox. Swing también soporta checkboxes en menús, utilizando la clase JCheckBoxMenuItem.

Como JCheckBoxMenuItem y JCheckBox descienden de AbstractButton, los checkboxes de Swing tienen todas las características de un botón normal.

Los checkboxes son similares a los botones de radio, pero su modelo de selección: ninguno, alguno o todos, pueden ser seleccionados. Sin embargo en un grupo de botones de radio, solo puede haber uno seleccionado.

Los métodos de AbstractButton que son mas utilizados son setMnemonic, addItemListener, setSelected y isSelected.

3.5.1 Métodos y constructores

Constructor	Propósito
JCheckBox(String) JCheckBox(String,boolean) JCheckBox(Icon) JCheckBox(Icon,boolean) JCheckBox(String,Icon) JCheckBox(String,Icon,boolean) JCheckBox()	Crea un ejemplar de JCheckBox. El argumento string especifica el texto, si existe, que el checkbox debería mostrar. De forma similar, el argumento Icon especifica la imagen que debería utilizarse en vez de la imagen por defecto del aspecto y comportamiento. Especificando el argumento booleano como true se inicializa el checkbox como seleccionado. Si el argumento booleano no existe o es false, el checkbox estará inicialmente desactivado.
JCheckBoxMenuItem(String) JCheckBoxMenuItem(String,boolean) JCheckBoxMenuItem(Icon) JCheckBoxMenuItem(String,Icon) JCheckBoxMenuItem(String,Icon, boolean) JCheckBoxMenuItem()	Crea un ejemplar de JCheckBoxMenuItem. Los argumentos se interpretan de la misma forma que en los constructores de JCheckBox.

3.5.2 Ejemplo

Utilizamos dos objetos de la clase JCheckBox para cambiar el estilo de la fuente del texto desplegado en un objeto JTextField. Uno aplica un estilo de negritas al seleccionarlo y el otro un estilo de cursivas. Ambos pueden aplicarse a la vez. De inicio ninguno esta seleccionado.

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

public class PruebaCasillaVerificacion extends JFrame {

private JTextField t;
private JCheckBox negrita, cursiva;
public PruebaCasillaVerificacion()
{

super( "Prueba de JCheckBox" );
Container c = getContentPane();
c.setLayout(new FlowLayout());

//creamos e inicializamos el objeto JTextField
t = new JTextField( "Texto que cambia", 28 );

//establecemos el estilo de la fuente y añadimos al panel
t.setFont( new Font( "TimesRoman",Font.PLAIN, 14 ) );
c.add( t );

// crea los objetos casilla de verificación

negrita = new JCheckBox( "Negrita" );
c.add( negrita );

cursiva = new JCheckBox( "Cursiva" );
c.add( cursiva );

/* si se hace click en un objeto JCheckBox se genera un ItemEvent que puede ser
     manejado por un ItemListener (cualquier objeto de una clase que implemente
la interfaz ItemListener) un objeto ItemListener debe definir el metodo
itemStateChanged */

ManejadorCasillaVerificacionmanejador = new ManejadorCasillaVerificacion();
negrita.addItemListener( manejador );
cursiva.addItemListener( manejador );


addWindowListener( new WindowAdapter() {
                        public void windowClosing( WindowEvent e ){
                              System.exit( 0 );
                        } // fin del método windowClosing
                  } // fin de la clase interna anónima
); // fin de addWindowListener


setSize( 325, 100 );
show();
} // fin del constructor de PruebaCasillaVerificacion


public static void main( String args[] ){
```

```
        new PruebaCasillaVerificacion();

}
```

```
private     class ManejadorCasillaVerificacion  implements
ItemListener {

private int valNegrita = Font.PLAIN;
private int valCursiva = Font.PLAIN;


/* con e.getSource() determinamos el onjeto sobre el que se hizo
Clic con las estructuras if-else se determina cual fue modificado y
la accion que tenemos que llevar a cabo */

      public void itemStateChanged( ItemEvent e )
      {

      if ( e.getSource() == negrita )
            if ( e.getStateChange() == ItemEvent.SELECTED )
                  valNegrita = Font.BOLD;
            else
                  valNegrita = Font.PLAIN;


      if ( e.getSource() == cursiva )
            if ( e.getStateChange() == ItemEvent.SELECTED )
                  valCursiva = Font.ITALIC;
            else
                  valCursiva = Font.PLAIN;


      //establecemos los tipos de la nueva fuente
      t.setFont(new Font( "TimesRoman", valNegrita + valCursiva, 14 ) );

      //repintamos el texto
      t.repaint();
      } // fin del método itemStateChanged

   } // fin de la clase interna //ManejadorCasillaVerificacion

} // fin de la clase PruebaCasillaVerificacion
```

3.6. *JRadioButton*

Los Botones de Radio son grupos de botones en los que, por convención, sólo uno de ellos puede estar seleccionado. Swing soporta botones de radio con las clases JRadioButton y ButtonGroup. Para poner un botón de radio en un menú, se utiliza la clase JRadioButtonMenuItem. Otras formas de presentar una entre varias opciones son los combo boxes y las listas.

Como JRadioButton desciende de AbstractButton, los botones de radio Swing tienen todas las caracterísitcas de los botones normales.Los métodos de AbstractButton que más se utilizan son setMnemonic, addItemListener, setSelected, y isSelected.

Para cada grupo de botones de radio, se necesita crear un ejemplar de ButtonGroup y añadirle cada uno de los botones de radio. El ButtonGroup tiene cuidado de desactivar la selección anterior cuando el usuario selecciona otro botón del grupo.

Generalmente se debería inicializar un grupo de botones de radio para que uno de ellos esté seleccionado. Sin embargo, la API no fuerza esta regla un grupo de botones de radio puede no tener selección inicial. Una vez que el usuario hace una selección, no existe forma para desactivar todos los botones de nuevo.

Cada vez que el usuario pulsa un botón de radio, (incluso si ya estaba seleccionado), el botón dispara un evento "action". También ocurren uno o dos eventos ítem: uno desde el botón que acaba de ser seleccionado, y otro desde el botón que ha perdido la selección (si existía). Normalmente, las pulsaciones de los botones de radio se manejan utilizando un oyente de "action".

3.6.1 Métodos y constructores

-ButtonGroups

Método	Propósito
ButtonGroup()	Crea un ejemplar de ButtonGroup.
void add(AbstractButton) void remove(AbstractButton)	Añade un botón a un grupo, o elimina un botón de un grupo

-RadioButton

Constructor	Propósito
JRadioButton(String) JRadioButton(String,boolean) JRadioButton(Icon) JRadioButton(Icon, boolean) JRadioButton(String, Icon) JRadioButton(String, Icon, boolean) JRadioButton()	Crea un ejemplar de JRadioButton. El argumento string especifica el texto, si existe, que debe mostrar el botón de radio. Similarmente, el argumento, Icon especifica la imagen que debe usar en vez la imagen por defecto de un botón de radio para el aspecto y comportamiento. Si se especifica trae en el argumento booleano, inicializa el botón de radio como seleccionado, sujeto a la aprovación del objeto ButtonGroup. Si el argumento booleano esta ausente o es false, el botón de radio está inicialmente deseleccionado.
JRadioButtonMenuItem(String) JRadioButtonMenuItem(Icon) JRadioButtonMenuItem(String, Icon)	Crea un ejemplar de JRadioButtonMenuItem. Los argumentos se interpretan de la misma forma que los de los constructores de JRadioButton.

3.6.2 Ejemplo

Según se seleccione uno u otro animal, se mostrara en la parte derecha de la ventana.

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

public class RadioButtonDemo extends JPanel {

static JFrame frame;
static String birdString = "Bird";
static String catString = "Cat";
static String dogString = "Dog";
static String rabbitString = "Rabbit";
static String pigString = "Pig";

JLabel picture;


public RadioButtonDemo() {


/* creamos los radio buttons el primero seleccionado es el birdButton tambien se usan
mnemonicos para acceso por teclado */

JRadioButton birdButton = new JRadioButton(birdString);

birdButton.setMnemonic('b');
birdButton.setActionCommand(birdString);
birdButton.setSelected(true);


JRadioButton catButton = new JRadioButton(catString); catButton.setMnemonic('c');
catButton.setActionCommand(catString);


JRadioButton dogButton = new JRadioButton(dogString); dogButton.setMnemonic('d');
dogButton.setActionCommand(dogString);


JRadioButton rabbitButton = new JRadioButton(rabbitString);
rabbitButton.setMnemonic('r'); rabbitButton.setActionCommand(rabbitString);


JRadioButton pigButton = new JRadioButton(pigString); pigButton.setMnemonic('p');
pigButton.setActionCommand(pigString);


//creamos un grupo y los anyadimos
ButtonGroup group = new ButtonGroup();
group.add(birdButton);
group.add(catButton);
group.add(dogButton);
group.add(rabbitButton);
group.add(pigButton);
```

```
//como en los ejemplos anteriores

RadioListener myListener = new RadioListener();

birdButton.addActionListener(myListener);
catButton.addActionListener(myListener); dogButton.addActionListener(myListener);
rabbitButton.addActionListener(myListener); pigButton.addActionListener(myListener);


/* utilizamos un objeto Icon como argumento para el constructor de JLabel de inico
mostraremos la imagen Bird.gif */

picture = new JLabel(new ImageIcon("images/"+ birdString + ".gif"));


//establecemos la dimension picture.setPreferredSize(new Dimension(177,122));


//establecemos un panel para la ordenacion de los componentes

JPanel radioPanel = new JPanel();
radioPanel.setLayout(new GridLayout(0, 1));
radioPanel.add(birdButton);
radioPanel.add(catButton);
radioPanel.add(dogButton);
radioPanel.add(rabbitButton);
radioPanel.add(pigButton);


/* con      BorderLayout      podemos      ordenar      los      componentes en      cinco
regiones: norte, sur ,este,oeste y centro  */

setLayout(new BorderLayout());

/* el panel se cargara en el oeste y la imagen se mostrara en el centro */

add(radioPanel, BorderLayout.WEST);
add(picture, BorderLayout.CENTER);


setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
}


//como en la creacion de los botones hicimos
//birdButton.setActionCommand(birdString);
//en todos los botones de radio, ahora sabemos que
//imagen tenemos que mostrar con
//e.getActionCommand()

class RadioListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {
            picture.setIcon(new ImageIcon("images/"
            + e.getActionCommand()+ ".gif"));

            }

}

public static void main(String s[]) {
```

```
        frame = new JFrame("RadioButtonDemo");
        frame.addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e)
                    {
                    System.exit(0);
                    }
              });

frame.getContentPane().add(new RadioButtonDemo(), BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);

}
}
```