1 Manejo de Errores con Exceptions

Cuando escribimos un programa en cualquier lenguaje de programación uno de los tópicos más importantes y más delicados es el manejo de errores. El momento ideal para capturar un error es en tiempo de compilación, es decir, antes de siquiera correr el programa. Sin embargo, no todos los errores pueden ser detectados en tiempo de compilación. Esto significa que varios de los errores de nuestros programas deben ser manejados en tiempo de ejecución, de tal forma que el desempeño de nuestro programa no se vea afectado por dichos errores.

Para solucionar este problema es necesario aislar la posible causa del error y "manejarla" de manera adecuada. El manejo de errores tiene una larga historia, su implementación viene desde los sistemas operativos en los años 60s en BASIC, Java se basa principalmente en el modelo de manejo de errores de C++. Al manejo de errores se le conoce como **exception handling**.

La palabra **exception** debe entenderse como *"ocurrió un error inesperado, no se cómo manejarlo, pero se que el programa no puede continuar"*.

En este capítulo aprenderemos el manejo de errores y su manejo.

Exceptions básicas.

Una condición de **exception** es un problema que impide la continuación de un método o de una parte de nuestro programa. El proceso no puede continuar porque no contamos con información suficiente para solucionar el problema surgido en el contexto actual.

Un ejemplo es una división. Si estamos a punto de dividir entre cero, sería conveniente asegurarnos de no continuar con el programa y o efectuar la operación. Como vemos, el valor "cero" es un caso excepcional el cual no esperábamos por lo que es más conveniente "***lanzar***" una excepción que continuar normalmente el programa.

Cuando lanzamos una excepción ocurren varias cosas. Primero, el objeto **exception** es creado de la misma manera que cualquier objeto en Java. Luego, el camino normal de ejecución del programa es cambiado hacia el mecanismo de "***manejo de excepciones***" para que el programa decida qué hacer con dicha excepción. Más adelante veremos en qué consiste dicho mecanismo.

Como ejemplo, consideremos una referencia a un objeto llamada **t**. Es posible que dicha referencia no sea inicializada por lo que deseamos verificar que su valor no sea nulo. Pero también deseamos enviar información del error generado, a dicha acción se le conoce como "***lanzar una excepción***(***throwing an exception***). El código de ejemplo es:

if(t==null)

```
throw new NullPointerException();
```

Argumentos de Exception.

Los constructores de las excepciones son dos: el primero es el constructor default que no recibe argumentos; el segundo toma como argumento una cadena **String** con información pertinente de la excepción:

```
if(t==null)
   throw new NullPointerException("t==null");
```

Típicamente, se lanzan excepciones diferentes para cada tipo diferente de error.

Capturando una Excepción.

Si una parte de nuestro programa lanza una excepción, debe asumirse que alguien debe "***capturar***" dicha excepción. Una de las ventajas del manejo de excepciones en Java es que podemos separar el código del problema que queremos resolver ,del código de los errores que se pueden generar.

Para entender como es atrapada una excepción, debemos entender primero el concepto de "***región segura***" (***guarded region***), que es una sección del código que puede producir excepciones.

Bloque try.

Si estamos dentro de un método y lanzamos una excepción, saldremos de dicho método debido a la excepción generada. Si no queremos que eso ocurra, podemos utilizar un bloque especial dentro del método para aislar la excepción. El bloque que vamos a utilizar es "**try**" y su sintaxis es:

```
try{
        // Código que puede generar excepciones
        }
```

De esta manera, dentro de un solo bloque podemos capturar las excepciones, aislándolas del resto del código de nuestro programa. Esto implica una cosa importante: ***el código es mucho más fácil de entender***.

Manejando excepciones.

Como mencionamos anteriormente, una vez que una excepción es lanzada debemos ser capaces de manejarla en algún otro lugar. Ese lugar es el "manejador de excepciones". Por cada excepción que se genere debe haber un bloque de este tipo. Los manejadores de excepciones se denotan con la palabra reservada **catch**:

```
try{
        // Código que puede generar excepciones
        } catch(Tipo1 id1){
                //Manejando excepciones Tipo1
        } catch(Tipo2 id2){
                //Manejando excepciones Tipo2
        } catch(Tipo3 id3){
                //Manejando excepciones Tipo3
                }
```

Cada bloque **catch** es como un pequeño método que tiene un solo argumento de un tipo particular. El identificador **id1, id2, etc**. Puede ser utilizado dentro del bloque, como el argumento de cualquier método.

Los manejadores deben aparecer después del bloque **try**. Si se genera una excepción , el mecanismo de manejo de excepciones busca el primer bloque catch con el argumento que concuerde con la excepción generada.

Termination y resumption.

Existen básicamente dos modelos en el manejo de excepciones. El modelo "**termination**" (que en que se basa Java y C++), asume que el error es tan crítico que no existe una manera de de regresar al lugar donde ocurrió la excepción.

La alternativa a este modelo es "resumption". En ella, el manejo de excepciones permite rectificar la situación y regresar a donde se generó la excepción. Como alternativa en Java, el bloque **try** puede estar dentro de un ciclo **while** que repita el bloque hasta que el resultado sea satisfactorio.

Aunque puede parecer que el segundo modelo es más atractivo, en la práctica resulta muy ineficiente. El código que se produce es demasiado de escribir y aún más difícil de entender.

Creando nuestras Excepciones.

Para crear nuestras propias clases de excepciones estamos obligados a heredar de alguna de las excepciones ya existentes, preferentemente alguna que se parezca a nuestra excepción. La manera más trivial de crear un nuevo tipo de excepción es dejar que el compilador cree el constructor predeterminado por nosotros, por lo que el código resulta muy corto:

```
//SimpleExcepcionDemo.java
//Creando excepciones.

class SimpleExcepcion extends Exception{}

public class SimpleExcepcionDemo{
        public void f() throws SimpleExcepcion{
                System.out.println("Lanzando Excepcion desde f()");
                throw new SimpleExcepcion();
                }
        public static void main(String[] args){
                SimpleExcepcionDemo sed=
                        new SimpleExcepcionDemo();
                try{
                        sed.f();
                        }catch(SimpleExcepcion e){
                                System.err.println("Atrapada!");
                        }
                }
        }
```

En este caso, el constructor es el predeterminado por el compilador. El código funciona de la siguiente manera: creamos un método **f()** que lanza una excepción. Dentro del método **main** observamos un bloque **try**, dado que sabemos que **f()** *"puede"* lanzar una excepción, aunque en este caso siempre lanza la excepción, debemos de asegurarnos que el código sea consistente. Finalmente aparece el bloque **catch** que maneja la excepción generada, en este caso, una excepción de tipo **SimpleExcepcion**.

Veamos un ejemplo donde el constructor tome un **String** como argumento:

```
//ContructorCompleto.java
//Creando nuestras propias excepciones.

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

```
public class ConstructorCompleto{
        public static void f() throws MiExcepcion{
                System.out.println("Lanzando excepciones desde f()");
                throw new MiExcepcion();
                }
        public static void g() throws MiExcepcion{
                System.out.println("Lanzando Excepciones desde g()");
                throw new MiExcepcion("Originada desde g()");
                }
        public static void main(String[] args){
                try{
                        f();
                        }catch(MiExcepcion e){
                                e.printStackTrace(System.err);
                        }
                try{
                        g();
                        }catch(MiExcepcion e){
                                e.printStackTrace(System.err);
                        }
                }
        }
```

Como se nota, el código agregado es realmente poco. El proceso de crear excepciones puede ir más allá:

```
//ExtraFeatures.java

class MiExcepcion2 extends Exception{
        public MiExcepcion2(){}
        public MiExcepcion2(String msg){
                super(msg);
                }
        public MiExcepcion2(String msg, int x){
                super(msg);
                i=x;
                }
        public int val(){return i;}
        private int i;
        }

public class ExtraFeatures{
        public static void f() throws MiExcepcion2{
                System.out.println("Lanzando MiExcepcion2 desde f()");
```

```
        throw new MiExcepcion2();
        }
public static void g() throws MiExcepcion2{
        System.out.println("Lanzando MiExcepcion2 desde g()");
        throw new MiExcepcion2("Originada desde g()");
        }
public static void h() throws MiExcepcion2{
        System.out.println("Lanzando Excepcion desde h()");
        throw new MiExcepcion2("Originada en h()",47);
        }
public static void main(String[] args){
        try{
                f();
                }catch(MiExcepcion2 e){
                        e.printStackTrace(System.err);
                }
        try{
                g();
                }catch(MiExcepcion2 e){
                        e.printStackTrace(System.err);
                }

        try{
                h();
                }catch(MiExcepcion2 e){
                        e.printStackTrace(System.err);
                }
        }
}
```

Como se vio en estos tres ejemplos, las excepciones pueden ser creadas como cualquier otro objeto. Por lo tanto podemos crear nuestras excepciones y construirlas como mejor nos convenga.

Especificando la existencia de Excepciones.

En Java, es obligatorio informar al cliente que utilice nuestras clases de la existencia de excepciones en dichos métodos. Dentro de la sintaxis de Java, existe una manera muy explícita de indicar esto, de hecho, en los ejemplos anteriores se estuvo utilizando. A este proceso se le conoce como especificación de excepciones y forma parte de la declaración de un método. Veamos un ejemplo:

```
void f() throws MuyGrande,MuyPequenio,DivCero{ //...
```

de esta forma indicamos que el método **f()** puede lanzar excepciones del tipo **MuyGrande**, **MuyPequenio** y/o **DivCero**.

Java no nos permite mentir acerca de las especificaciones de excepciones. Si nuestro método puede lanzar alguna excepción, entonces el compilador nos **OBLIGARÁ** a indicarlo explícitamente.

Atrapando cualquier Excepción.

Es posible crear un bloque **catch** capaz de atrapar cualquier tipo de excepción. Esto se puede lograr atrapando a la clase base **Exception** como en el siguiente ejemplo:

```
catch(Exception e){
        System.err.println("Atrape una excepcion");
        }
```

Ya que **Exception** es la clase base de todas las excepciones, no se obtiene mucha información acerca de la excepción, pero podemos llamar cualquiera de los métodos de su tipo clase padre **Throwable**:

```
String getMessage()
String getLocalizedMessage()
String toString()
void printStackTrace()
void printStackTrace(PrintStream)
void printStackTrace(PrintWriter)
Throwable fillInStackTrace()
```

Veamos un ejemplo de los métodos:

```
//MetodosException.java
//Demostrando el uso de los métodos.

public class MetodosException{
        public static void main(String[] args){
                try{
                        throw new Exception("Aqui esta mi Excepcion");
                        }catch(Exception e){
                                System.err.println("Excepcion atrapada");
                                System.err.println(
                                        "e.getMessage():"+e.getMessage());
                                System.err.println(
                "e.getLocalizedMessage():"+e.getLocalizedMessage());
                                System.err.println(
                                        "e.toString():"+e);
                                System.err.println(
```

```
                "e.printStackTrace():");
                        e.printStackTrace(System.err);
                }
        }
}
```

Excepciones Estándar de Java.

La clase de Java Throwable describe todo lo que puede ser lanzado en una excepción. Existen dos tipos de objetos Throwable: Error representa los errores de tiempo de compilación y no son necesarios atraparlos; Exception es la clase base y ocurren en tiempo de ejecución.

La mejor manera de revisar la lista de excepciones es en la página java.sun.com. El número de excepciones en Java continúa expandiéndose, por lo que resulta inútil listarlas.

Caso especial: RuntimeException.

El primer ejemplo de esta sección fue:

```
if(t==null)
    throw new NullPointerException();
```

Puede resultar bastante molesto verificar cada referencia que hagamos. Afortunadamente no tenemos que realizar dicha acción ya que Java la realiza por nosotros. Este tipo de excepciones especiales reciben el nombre de RuntimeException. Son lanzadas automáticamente por Java y no necesitamos incluirlas en nuestro código explícitamente. Veamos un ejemplo:

```
//NuncaAtrapada.java
//Ignorando RuntimeExceptions

public class NuncAtrapada{
        static void f(){
                throw new RuntimeException("Desde f()");
                }
        static void g(){
                f();
                }
        public static void main(String[] args){
                g();
```

```
        }
    }
```

Como vemos, las excepciones de tipo **RuntimeException** que son lanzadas fuera de **main**, automáticamente llaman a **printStackTrace**().

Existen diversas razones por las cuales existan excepciones **RuntimeException**:

1. Un error que no podemos atrapar (por ejemplo, recibir una referencia de algún objeto no inicializado).
2. Un error como programador, es decir, un error en la lógica o secuencia de nuestro programa.

Restricciones de Excepciones.

Cuando sobrescribimos un método, podemos sólo podemos lanzar excepciones que hayan sido especificadas en la clase base de dicho método. Veamos un ejemplo:

```
//InningTormentoso.java
//Métodos sobrescritos sólo pueden lanzar excepciones especificadas
//en los métodos de la clase padre.

class BaseballExcepcion extends Exception{}
class Foul extends BaseballExcepcion{}
class Strike extends BaseballExcepcion{}
abstract class Inning{
        Inning() throws BaseballExcepcion{}
        void event() throws BaseballExcepcion{
                //No tiene porque hacer algo
                }
        abstract void alBat() throws Strike,Foul;
        void walk(){} //No lanza nada.
        }
class TormentosaExcepcion extends Exception{}
class RainedOut extends TormentosaExcepcion{}
class PopFoul extends Foul{}

interface Tormenta{
        void event() throws RainedOut;
        void rainHard() throws RainedOut;
        }

public class InningTormentoso extends Inning implements Tormenta{
        //Es correcto agregar nuevas excepciones a los constructors
        //pero es necesario manejar las de la clase base.
```

```
        InningTormentoso() throws RainedOut,BaseballExcepcion{}
        InningTormentoso(String s) throws Foul,BaseballExcepcion{}
        //Métodos ordinarios deben ser idénticos a los de la clase padre
        //!void walk() throws PopFoul{}
        //Las interfaces NO PUEDEN agregar excepciones a los métodos
        //existentes
        //!public void event() throws RainedOut{}
        //Si el método es  idéntico en la clase padre, la excepción esta
//bien
        public void rainHard() throws RainedOut{}
        public void event(){}
        //Los métodos sobrescritos pueden lanzar excepciones heredadas.
        void alBat() throws PopFoul{}
        public static void main(String[] args){
                try{
                        InningTormentoso si=new InningTormentoso();
                        si.alBat();
                        }catch(PopFoul e){
                                System.err.println("Pop foul");
                        }catch(RainedOut e){
                                System.err.println("Rained out");
                        }catch(BaseballExcepcion e){
                                System.err.println("Error Generico");
                        }
                try{
                        Inning i=new InningTormentoso();
                        i.alBat();
                        //debes atrapar todas las excepciones de la clase
                //padre
                        }catch(Strike e){
                                System.err.println("Strike");
                        }catch(Foul e){
                                System.err.println("Foul");
                        }catch(RainedOut e){
                                System.err.println("Rained out");
                        }catch(BaseballExcepcion e){
                                System.err.println("Excepcion generica");
                        }
                }
        }
```

Más sobre excepciones.

Cuando una excepción es lanzada, el sistema de manejo de excepciones busca el bloque catch más cercano. Cuando lo encuentra, la excepción se considera manejada y no ocurre nada más. Esto puede implicar que el manejador no sea el adecuado:

```
//Humano.java
//Jerarquía de excepciones.

class Molesto extends Exception{}
class Enfermo extends Molesto{}
public class Humano{
        public static void main(String[] args){
                try{
                        throw new Enfermo();
                        }catch(Enfermo e){
                                System.err.println("Atrapado Enfermo");
                        }catch(Molesto e){
                                System.err.println("Atrapado Molesto");
                        }
                }
        }
```

La excepción Enfermo será atrapada por el primer bloque catch. Sin embargo si el código fuera:

```
class Molesto extends Exception{}
class Enfermo extends Molesto{}
public class Humano{
        public static void main(String[] args){
                try{
                        throw new Enfermo();
                        }catch(Molesto e){
                                System.err.println("Atrapado Molesto");
                        }
                }
        }
```

el programa seguirá funcionando porque **Enfermo** es clase derivada de **Molesto**. Dicho de otra manera: **catch(Molesto e)** atrapará cualquier **Molesto** o clase derivada de ella.

Guía de Excepciones.

Finalmente veamos una guía de cuando utilizar las excepciones:

1. Arreglar un problema mas o menos previsto dentro de un método.
2. No arreglar el problema y simplemente pasarlo por alto.
3. Hacer algo alternativo a lo que el método iba a realizar.
4. Terminar el programa.

2 El Sistema I / O de Java.

Una de las tareas más difíciles en el diseño de cualquier lenguaje de programación es crear un buen sistema de entrada / salida (I/O). Los diseñadores de la librería de Java atacaron el problema creando una gran cantidad de clases. De hecho, son tantas clases que puede resultar intimidante al principio. Además, desde la primera versión de Java, el sistema I/O ha evolucionado para nuestra comodidad. Es importante entender la evolución de dicho sistema ya que al entender dicha perspectiva, será más fácil adaptarnos a las clases existentes y saber cuándo debemos usar dichas clases y cuándo no.

La Clase File.

Esta clase representa el nombre de un archivo en particular o bien, los nombres de una lista de archivos en un directorio. En esta sección, veremos ejemplos de uso de dicha clase:

Un listado de un Directorio.

Supongamos que queremos realizar un listado de un directorio. El objeto **File** puede utilizarse en dos maneras. Si usamos el método **list**() sin argumentos, obtendremos la lista completa de lo que contiene el objeto **File**. Sin embargo, si queremos una lista más selectiva, podremos utilizar un ***filtrado***. Veamos un ejemplo:

```
//DirList.java
//Muestra un listado de un directorio.

import java.io.*;
import java.util. Arrays ;
import com.serge.util..*;

public class DirList{
        public static void main(String[] args){
                File path=new File(".");
                String[] lista;
                if(args.length==0)
                        lista=path.list();
                else
                        lista=path.list(new DirFilter(args[0]));
                Arrays.sort(lista,new ComparadorAlfabetico());
                for(int i=0;i<lista.length;i++)
                        System.out.println(lista[i]);
                }
        }
class DirFilter implements FilenameFilter{
```

```
String afn;
DirFilter(String afn){ this.afn=afn; }
public boolean accept(File dir,String nombre){
        String f=new File(nombre).getName();
        return f.indexOf(afn) !=-1;
        }
}
```

La clase DirFilter implementa la interfaz FilenameFilter, veamos lo simple que es la interfaz:

```
public interface FilenameFilter{
        boolean accept(File dir, String nombre);
        }
```

El método **accept()** debe aceptar un objeto **File** el cual representa el directorio donde se encuentra un archivo en particular, y una cadena con el nombre de dicho archivo. El método **list()** es el encargado de llamar al método **accept()** para cada una de los archivos en el directorio.

Veamos otro ejemplo:

```
//DirList2.java

import java.io.*;
import java.util.Arrays;
import com.serge.util.*;

public class DirList2{
        public static FilenameFilter filter(final String afn){
                //clase interna
                return new FilenameFilter(){
                        String fn=afn;
                        public boolean accept(File dir,String n){
                                String f=new File(n).getName();
                                return f.indexOf(fn)!= -1;
                                }
                        };//Fin de la clase interna
                }
        public static void main(String[] args){
                File path=new File(".");
                String[] lista;
                if(args.length==0)
                        lista=path.list();
                else
                        lista=path.list(filter(args[0]));
```

```
            Arrays.sort(lista,new ComparadorAlfabetico());
            for(int i=0;i<lista.length;i++)
                  System.out.println(lista[i]);
            }
}
```

El único cambio en este ejemplo con respecto al anterior es que tomamos la clase **FilenameFilter** directamente sin necesidad de tener que implementarla, esto se logra mediante el uso de clases internas.

Creando directorios.

Con la clase File podemos crear nuevos directorios o la ruta de un nuevo directorio si éste no existe. También podemos observar las características de los archivos como son tamaño, lectura, escritura, etc. El siguiente ejemplo muestra algunos métodos de la clase File.

```
//ClaseFile.java
//Muestra el uso de la clase File para crear directorios y manipular
//archivos

import java.io.*;

public class ClaseFile{
      private final static String uso=
            "Uso:CreandoDirectorios path1…\n"+
            "Creando cada path\n"+
            "Uso:CreandoDirectorios – d path…\n"+
            "Borrando cada path\n"+
            "Uso:CreandoDirectorios –r path1 path2\n"+
            "Renombrando el path1 en path2\n";
      private static void uso(){
            System.err.println(uso);
            System.exit(1);
            }
      private static void fileData(File f){
            System.out.println("Path absoluto: +"f.getAbsolutePath()+
                              "\n Lectura: " +f.canRead()+
                              "\n Escritura: "+f.canWrite()+
                              "\n Nombre: "+f.getName()+
                              "\n Padre: " +f.getParent()+
                              "\n Path: "+f.getPath()+
                              "\nlongitud: "+f.length()+
                              "\nModificacion? :"+f.lastModified());
            if(f.isFile())
```

```
                System.out.println("Es un archivo");
        else
                System.out.println("Es un directorio");
        }
public static void main(String[] args){
        if(args.length<1) uso();
        if(args[0].equals("-r")){
                if(args.length!=3) uso();
                File
                        viejo=new File(args[1]),
                        rname=new File(args[2]);
                viejo.renameTo(rname);
                fileData(viejo);
                fileData(rname);
                return; //regresar a main
                }
        int contador=0;
        boolean del=false;
        if(args[0].equals("-d")){
                contador++;
                del=true;
                }
        for(;contador<args.length;contador++){
                File f=new File(args[contador]);
                if(f.exists()){
                        System.out.println(f+"Existe");
                        if(del){
                                System.out.println("Borrando…"+f);
                                f.delete();
                                }
                        }
                else{//no existe
                        if(!del){
                                f.mkdirs();
                                System.out.println("Creado"+f);
                                }
                        }
                fileData(f);
                }
        }
}
```

Input y Output.

Las clases de Java se dividen para el sistema de I/O en dos: input y output. Por herencia, todo aquello que se derive de las clases **InputStream** o **Reader** tendrán los métodos básicos llamados **read**() para lectura simple de bytes o arreglos de bytes. De manera similar, todo aquello que derive de las clases **OutputStream** o **Writer** tendrán los métodos básicos **write**() para escritura de bytes o arreglo de bytes. Sin embargo, generalmente no usaremos dichos métodos ya que existen otras clases que nos permiten trabajar de manera más cómoda.

Debe de quedar claro que todo lo que tiene que ver con un flujo de información de entrada o **Input** se maneja a través de la herencia de la clase **InputStream**; a su vez, todo lo que tiene que ver con la salida o **output**, se maneja con las clases asociadas a **OutputStream**.

Tipos de flujos de Entrada.

El trabajo de InputStream es representar a las clases que producen una entrada de cualquiera de los siguientes tipos:

1. Un arreglo de bytes.
2. Un objeto **String**.
3. Un archivo.
4. Un "**conducto**" o "**pipe**" que es un camino que comienza en un lugar para llevar la información a un destino final.
5. Una secuencia de flujos para almacenarlos en uno sólo.
6. Otras fuentes, como la propia Internet.

Cada uno de estos tipos tiene una subclase asociada con **InputStream**. Además, **FilterInputStream** también es un tipo de **InputStream**. Esto se mostrará más adelante. La siguiente tabla presenta las clases asociadas.

Clase	Función	Argumentos del constructor / Cómo usarla
ByteArrayInputStream	Permite que lo que está almacenado en el buffer sea usado en como un **InputStream**	El buffer de donde se extraen los bytes
		Como fuente de datos. Conectarlo a un objeto **FileInputStream** para tener una útil interfaz.

StringBufferInputStream	Convierte un **String** en un **InputStream**	Un **String**. También podemos utilizar un **StringBuffer**
		Como fuente de datos. Conectarlo a un objeto **FilterInputStream** para tener una útil interfaz.
FileInputStream	Para leer información de un archivo	Un **String** que representa el nombre de un archivo, o un objeto **File** o **FileDescriptor**.
		Como fuente de datos. Conectarlo a un objeto **FiterInputStream** para tener una interfaz útil.
PipedInputStream	Produce la información escrita al objeto asociado **PipedOutputStream**.	**PipedOutputStream**
		Como fuente de datos. Conectarlo a un objeto **FiterInputStream** para tener una interfaz útil.
SequenceInputStream	Convierte dos o más objetos **InputStream** en un solo **InputStream**	Dos objetos **InputStream** o un objeto **Enumeration**.
		Como fuente de datos. Conectarlo a un objeto **FiterInputStream** para tener una interfaz útil.
FilterInputStream	Clase abstracta cuya interfaz provee funcionalidad a las clases **InputStream**	

Tipos de Flujos de Salida.

Esta categoría incluye las clases que definen a dónde irán los flujos de salida: un arreglo de bytes, un archivo o un *"**pipe**"*.

Clase	**Función**	**Argumentos del constructor** **Cómo usarla**
ByteArrayOutputStream	Crea un buffer en la memoria. Toda la información que se le mande será almacenada en el buffer.	Opcionalmente el tamaño inicial del buffer. Para designar el destino de nuestra información. Conectarlo a un objeto **FilterOutputStream** para obtener una interfaz útil.
FileOutputStream	Para enviar información de a un archivo.	Un **String** que representa el nombre de un archivo, o un objeto **File** o **FileDescriptor**. Para designar el destino de nuestra información. Conectarlo a un objeto **FilterOutputStream** para obtener una interfaz útil.
PipedOutputStream	Cualquier información escrita a la salida de este objeto será automáticamente la entrada del objeto asociado **PipedInputStream**.	**PipedInputStream** Para designar el destino de nuestra información. Conectarlo a un objeto FilterOutputStream para obtener una interfaz útil.

FilterOutputStream	Clase abstracta cuya interfaz provee funcionalidad a las clases **OutputStream**	

Las clases FilterOutputStream y FilterInputStream son clases abstractas que se derivan directamente de las clases bases de la librería I/O, InputStream y OutPutStream, y proveen una interfaz común a todos los objetos derivados de estas clases.

Leyendo de un InputStream con FilterInputStream.

Las clases **FilterInputStream** cumplen con dos objetivos importantes. **DataInputstream** permite leer diferentes tipos de datos primitivos así como objetos **String**. Esto se complementa con la clase **DataOutputStream** que permite mover datos primitivos de un lugar a otro a través de un flujo. La siguiente tabla muestra los tipos de **FilterInputStream**.

Clase	Función	Argumentos del constructor / Cómo usarla.
DataInputStream	Usada en conjunto con **DataOutputStream** para leer datos primitivos.	**InputStream** Contiene una completa interfaz que permite leer datos primitivos.
BufferedInputStream	Usada para prevenir una lectura física cada vez que se desea alguna información. En pocas palabras “usando un buffer”.	**InputStream** con el tamaño del buffer como dato opcional. Es un requerimiento para usar un buffer.
LineNumberInputStream	Vigila el número de líneas del flujo de entrada.	**InputStream** Sólo nos informa del número de líneas.
PushbackInputStream		**InputStream**. Usado por el compilador. Probablemente no tenga un uso de aplicación.

Escribiendo a un OutputStream con FilterOutputStream.

El complemento a DataInputStream es DataOutputStream. Todos los métodos de estas clases comienzan con la palabra write como writeByte(), writeFloat(), etc.

El objetivo de DataOuputStream es poner los datos en un flujo de manera que DataInputStream sea capaz de reconstruirlos.

La clase PrintStream puede resultar problemática ya que atrapa todas las IOExceptions.

La clase BufferedOutputStream es un moificador que le dice al flujo que utilice un buffer para no utilizar escritura física cada vez que escribimos. Veamos la tabla correspondiente:

Clase	Función	Argumentos del constructor Cómo usarla.
DataOutputStream	Usada en conjunto con **DataInputStream** para escribir datos primitivos.	**OutputStream** Contiene una completa interfaz que permite escribir datos primitivos.
BufferedInputStream	Usada para prevenir una lectura física cada vez que se desea enviar información. En pocas palabras "usando un buffer".	**OutputStream** con el tamaño del buffer como dato opcional. Es un requerimiento para usar un buffer.
PrintStream	Para producir salida con formato.	**OutputStream** con la opción de indicar si de vaciar el buffer cada nueva línea Debe ser la última línea de un objeto OutputStream. Se utiliza con frecuencia.

*Reader*s & *Writer*s

Java 1.1 realizó significativos avances a la librería de I/O. Cuando vemos las clases **Reader** y **Writer** podemos pensar que reemplazan a las antiguas clases **InputStream** y **OutputStream**. Sin embargo, las clases **InputStream** y **OutputStream** proveen mucha funcionalidad en el manejo de bytes de I/O, mientras que las clases **Reader** y **Writer** son

orientadas a ***Unicode***, que es un código basado en caracteres. Además Java 1.1 agregó clases a **InputStream** y **OutputStream**, clases que nos permiten combinar el uso de bytes con caracteres como **InputStreamReader** y **OutputStreamReader**.

El uso más importante de las clases **Reader** y **Writer** es la internacionalización. El viejo sistema de I/O soporta 8 bit y no maneja el código de 16 bit de **Unicode**.

Casi todas las clases de flujos originales de I/O tienen sus correspondientes clases en **Reader** y **Writer**. Veamos la tabla de correspondencias:

Java 1.0	Java 1.1
InputStream	**Reader** **Convertidor:** **InputStreamReader**
OutputStream	**Writer** **Convertidor:** **OutputStreamReader**
FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(no tiene correspondencia)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

Así mismo la interfaz **FilterInputStream** tiene una "*similitud*" en las nuevas clases. Existen diferencias en estos casos, ya que mientras que **BufferedOutputStream** es una subclase de **FilterOutputStream**, **BufferedWriter** no es una subclase de **FilterWriter**. Veamos la tabla:

Filtros Java 1.0	Java 1.1
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter (clase abstracta sin subclases)**
BufferedInputStream	**BufferedReader**
BufferedOutputStream	**BufferedWriter**
DataInputStream	**DataReader**
PrintStream	**PrintWriter**
LineNumberInputStream	**LineNumberReader**
StreamTokenizer	**StreamTokenizer**
PushBackInputStream	**PushBackReader**

Algunas clases no cambiaron de Java 1.0 a Java 1.1:

Clases que no cambiaron de Java 1.0 a Java 1.1
DataOutputStream **File**
RandomAccesFile
SequenceInputStream.

RandomAccesFile.

La clase **RandomAccesFile** es usada en archivos que contienen información de tamaño predeterminado de tal manera que se puede mover esa información de un lugar a otro con el método **seek**().

Ejemplo de aplicación de flujos de I/O.

El siguiente ejemplo puede observarse como referencia básica de los usos de I/O.

```
//IOStreamDemo.java
//Usos tipicos de IO.
import java.io.*;

public class IOStreamDemo{
        public static void main(String[] args) throws IOException{
                //1.-Leyendo la entrada por lineas
                BufferedReader in=new
                BufferedReader(new FileReader("IOStreamDemo.java"));
                String s,s2=new String();
                while((s=in.readLine())!=null)
                        s2!=s="\n";
                in.close();

                //1.b Leyendo la entrada estandar
                BufferedReader stdin=new BufferedReader(
                                        new InputStreamReader(System.in));
                System.out.print("Teclea una linea: ");
                System.out.println(stdin.readLine());

                //2 Memoria de entrada
                StringReader in2=new StringReader(s2);
                int c;
                while((c=in2.read())!= -1)
                        System.out.print((char)c);
```

```
//3. Entrada de memoria formateada
try{
        DataInputStream in3= new DataInputStream(
        new    ByteArrayInputStream(s2.getBytes()));
        while(true)
                System.out.print((char)in3.readByte());
                }catch(EOFException e){
                        System.err.println("Fin de Flujo");
        }

//4. Salida de Archivo
try{
        BufferedReader in4=new BufferedReader(
                        new StringReader(s2));
        PrintWriter out1=new PrintWriter(
                        new BufferedWriter(
                        new FileWriter("IODemo.out")));
        int cuentaLinea=1;
        while((s=in4.readLine())!=null)
                out1.println(cuentaLinea++ +": "+s);
        out1.close();
        }catch(EOFException e){
                System.err.println("Fin de Flujo");
        }

//5. Almacenando y recuperando datos
try{
        DataOutputStream out2=new DataOutputStream(
                new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
        out2.writeDouble(3.14159);
        out2.writeChars("Eso fue pi\n");
        out2.writeBytes("Eso fue pi\n");
        out2.close();
        DataInputStream in5=new DataInputStream(
                new BufferedInputStream(
                        new FileInputStream("Data.txt")));
        BufferedReader in5br=new BufferedReader(
                new InputStreamReader(in5));
        //Debemos usar DataInputStream para datos
        System.out.println(in5.readDouble());
        //Ahora podemos usar el readLine apropiado
        System.out.println(in5br.readLine());
        System.out.println(in5br.readLine());
        }catch(EOFException e){
                System.err.println("Fin de flujo");
```

```
        }

        //6. Leyendo/Escribiendo random acces files
        RandomAccessFile rf=
                new RandomAccessFile("rtest.dat","rw");
        for(int i=0;i<10;i++)
                rf.writeDouble(i*1.414);
        rf.close();
        rf=new RandomAccessFile("rtest.dat","rw");
        rf.seek(5*8);
        rf.writeDouble(47.0001);
        rf.close();

        rf=new RandomAccessFile("rtest.dat","r");
        for(int i=0;i<10;i++)
                System.out.println("Valor"+i+": "
                        +rf.readDouble());
        rf.close();
        }
}
```

Veamos las secciones del código del programa tal como están marcadas:

1.- Entrada de archivo Buffered

Para abrir un archivo con entrada de caracteres, usamos un **FileInputReader** con un **String** o un objeto **File** como el nombre del archivo. Queremos que dicho archivo se trate como un buffer entonces la referencia resultante es el argumento del constructor de **BufferedReader**. Cuando leemos el final del archivo, **readLine**() regresa **null** entonces rompemos el ciclo **while**.

La cadena **s2** es utilizada para acumular el contenido entero del archivo. Finalmente, **close**() cierra la conexión con el archivo.

La sección 1b muestra cómo podemos leer **System.in. System.in** es un **DataInputStream** y **BufferedReader** necesita un objeto **Reader** como argumento, por lo que utilizamos **InputStreamReader** para transformarlos.

2.- Entrada desde la memoria.

Esta sección toma la cadena **s2** que contiene todo el archivo que leímos anteriormente y lo utiliza para crear un **StringReader**. El método **read**() regresa el próximo byte como un entero por lo que debemos hacer un ***cast*** a **char** para que se imprima correctamente.

3.- Entrada de la memoria formateada.

Para leer datos "con formato", se utiliza **DataInputStream**, ya que es una clase orientada a bytes. Así que debemos utilizar todas las clases de tipo **InputStream** en lugar de las **Reader**. Para convertir la cadena a un arreglo de bytes utilizamos el método **getBytes**() de **String**, ya que **ByteArrayStreamInput** toma como argumento un arreglo de bytes. Cuando leemos carácter por carácter usando el método **readByte**(), cualquier byte es válido por lo que no podremos detectar el fin de la entrada. Por lo cual podemos utilizar el método **available**() para saber cuántos caracteres quedan por leer.

```
//TestEOF.java
//Probando el fin de un archivo
//leyendo byte por byte.
import java.io.*;
public class TesEOF{
        //Lanza Excepciones
        public static void main(String[] args) throws IOException{
                DataInputStream din=new DataInputStream(
new BufferedInputStream(new FileInputStream("TestEOF.java")));
                while(din.available()!=0)
                        System.out.println((char)din.readByte());
                }
        }
```

4.- Salida del Archivo.

Este ejemplo también muestra cómo leer datos. Primero, **FileWriter** es creado para almacenar al archivo. Siempre es aconsejable utilizar **BufferedWriter** para almacenar en el buffer. El formato del archivo se da con el objeto **PrintWriter**.

De nuevo, cuando la fuente que nos proporciona la información se acaba, el método **readLine**() devuelve ***null***. También cerramos el flujo con el método **close**().

5.- Almacenando y recuperando Información.

Un objeto PrintWriter puede dar formato a la salida. Sin embargo, para el uso de bytes utilizamos las clases DataInputStream y DataOutputStream. La cadena es utilizada para escribir con los métodos writeChars() y writeBytes().

6.- Leyendo y escribiendo con random access file.

Aunque **RandomAccessFile** esté totalmente asilada de las demás clases de I/O , implementa las interfaces **DataInput** y **DataOutput**.

Ahora veamos otro ejemplo:

```
//IOProblem.java
import java.io.*;
public class IOProblem{
        public static void main(String[] args)throws IOException{
                DataOutputStream dos=new DataOutputStream(new
                                                BufferedOutputStream(
                                        new FileOutputStream("Data.txt")));
                dos.writeDouble(3.14159);
                dos.writeBytes("Ese era el valor de pi");
                dos.writeBytes("Este es pi/2: \n");
                dos.writeDouble(3.14159/2);
                dos.close();
                DataInputStream dis=new DataInputStream(new
                                                BufferedInputStream(
                                        new FileInputStream("Data.txt")));
                BufferedReader inbr=new BufferedReader(new
                                                InputStreamReader(dis));
                System.out.println(in.readDouble());
                System.out.println(inbr.readLine());
                System.out.println(inbr.readLine());
                System.out.println(in.readDouble());
                }
        }
```

Leyendo de la entrada Estándar.

Java cuenta con System.in, System.out y System.err. A través de los ejemplos hamos visto el uso de System.out y System.err, sin embargo, la entrada estándar no es usualmente utilizada. Para hacerlo debemos utilizar un buffer y convertir System.in a un objeto Reader.

```
//Echo.java
//Como leer desde la entrada estandar.

import java.io.*;
public class Echo{
        public static void main(String[] args)throws IOException{
                BufferedReader in=new BufferedReader(new
                                        InputStreamReader(System.in));
                String s;
                while((s=in.readLine()).length()!=0)
                        System.out.println(s);
                }
        }
```

También podemos cambiar el sistema **System.out**, ya que es un **PrintWriter**, el cual toma como argumento un objeto **OutputStream**.

```
//CambiandoSystemOut.java
//Cambiando la salida estandar
import java.io.*;
public class CambiandoSystemOut{
        public static void main(String[] args){
                PrintWriter out=new PrintWriter(System.out,true);
                out.println("Hola mundo");
                }
        }
```

Redireccionando I/O.

El sistema Java permite redireccionar la salida, error y entrada estándar usando llamadas a métodos simples:

```
setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)
```

Esto resulta especialmente útil si creamos sistemas que envíen mucha información a la pantalla por ejemplo.

```
//Redireccionando.java
import java.io.*;
public class Redireccionando{
        public static void main(String[] args)throws IOException{
                BufferedInputStream in=new BufferedInputStream(
                        new FileInputStream("Redireccionando.java"));
        PrintStream out=new PrintStream(new BufferedOutputStream(
                                        new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);

        BufferedReader br=new BufferedReader(new
                                InputStreamReader(System.in));
        String s;
        while((s=br.readLine())!=null)
                System.out.println(s);
        out.close();//siempre hay que hacerlo!
        }
```

```
}
```

Compresión.

La librería I/O contiene clases que permiten la lectura y escritura de flujos que están en formato comprimido. Estas clases no se derivan de **Reader** o **Writer** sino de **InputStream** y **OutputStream**.

Veamos la siguiente tabla:

Clase	Función
CheckedInputStream	**GetCheckSum**() produce checksum para cualquier objeto **InputStream**.
CheckedOutputStream	**GetCheckSum**() produce checksum para cualquier OutputStream.
DeflaterOutputStream	Clase base para la compresión.
ZipOutputStream	Un objeto **DeflaterOutputStream** que comprime datos en formato ***Zip***
GZIPOutputStream	Un objeto **DeflaterOutputStream** que comprime datos en formato ***Zip***
InflaterInputStream	Clase base para la decompresión.
ZipInputStream	Un objeto **InflaterInputStream** que descomprime la información en formato ***Zip***
GZIPInputStream	Un objeto **InflaterInputStream** que descomprime la información en formato ***GZIP***.

Compresión con GZIP.

La interfaz de GZIP es simple y más apropiada cuando tenemos un flujo simple de datos.

```
//GZIP.java
import java.io.*;
import java.util.zip.*;
public class GZIP{
        public static void main(String[] args)throws IOException{
                BufferedReader in=new BufferedReader(new
                        FileReader(args[0]));
                BufferedOutputStream out=new BufferedOutputStream(
                        new GZIPOutputStream(new FileOutputStream(
                                "test.gz")));
                System.out.println("Escribiendo el archive");
```

```
                int c;
                while((c=in.read())!=-1)
                        out.write( c );
                in.close();
                out.close();
                System.out.println("Leyendo el archivo");
                BufferedReader in2=new BufferedReader(new
                        InputStreamReader(new GZIPInputStream(
                                new FileInputStream("test.gz"))));
                String s;
                while((s=in2.readLine())!=null)
                        System.out.println(s);
                }
        }
```

El uso de las clases de compresión es bastante directo, simplemente la salida es un GZIPOutputStream o un ZipOutpitStream y la entrada es un GZIPInputStream o un ZipInputStream.

Almacenamiento múltiple con ZIP.

La librería que soporta Zip es mucho más completa. Podemos almacenar fácilmente múltiples archivos y aun así resulta sencillo el proceso de leerlos después. El siguiente ejemplo tiene la misma forma que el ejemplo anterior, pero toma como argumentos la línea de comandos para formar el archivo ZIP.

```
//ZIP.java
import java.io.*;
import java.util.zip.* ;
import java.util.* ;

public class ZIP{
        public static void main(String[] args)throws IOException{
                FileOutputStream f=new FileOutputStream("test.zip");
                CheckedOutputStream csum=
                        new CheckedOutputStream(f, new Adler32());
                ZipOutputStream out=new ZipOutputStream(
                                new BufferedOutputStream(csum));
                out.setComment("Prueba de Java Zip");
                for(int i=0 ;i<args.length ;i++){
                        System.out.println("Escribiendo archivo "+args[i]);
                        BufferedReader in=new BufferedReader(
                                        new FileReader(args[i]));
                        out.putNextEntry(new ZipEntry(args[i]));
```

```
                        int c;
                        while((c=in.read())!= -1)
                                out.write( c );
                        in.close();
                        }
                out.close();
                System.out.println("Checksum: "
+csum.getChecksum().getValue());
                System.out.println("Leyendo el archivo");
                FileInputStream fi=new FileInputStream("test.zip");
                CheckedInputStream csumi=
                        new CheckedInputStream(fi, new Adler32());
                ZipInputStream in2=ne w ZipInputStream(
                                        new BufferedInputStream(csumi));
                ZipEntry ze;
                while((ze=in2.getNextEntry())!=null){
                        System.out.println("Leyendo el archivo "+ze);
                        int x;
                        while((x=in2.read())!=-1)
                                System.out.write(x);
                        }
                System.out.println("CheckSum: "
                                        +csumi.getChecksum().getValue());
                in2.close();

                //Manera alternativa de hacer zip archivos
                ZipFile zf=new ZipFile("test.zip");
                Enumeration e=zf.entries();
                while(e.hasMoreElements()){
                        ZipEntry ze2=(ZipEntry)e.nextElement();
                        System.out.println("Archivo: "+ze2);
                        //y extraemos la información como en arriba!
                        }
                }
        }
```

Para cada archivo que deseamos agregar hacemos una llamada a **putNextEntry**() y pasarlo a **ZipEntry**. El objeto **ZipEntry** contiene una interfaz que permite obtener y escribir nombre del archivo ZIP, comentarios, método de compresión entre otros.

Para extraer los archivos, **ZIPInputStream** tiene el método **getNextEntry**() que regresa el objeto **ZIPEntry** si todavía existe otro archivo.

Archivos Java JARS.

El formato ZIP es también utilizado en los llamados **JAR** (“Java Archives”) . Los **JAR** son una manera de almacenar varios archivos en uno solo, de manera similar al método ZIP. Sin embargo, los archivos Java son compatibles en todas las plataformas.

Los archivos **JAR** son particularmente utilizados en Internet. Antes de la existencia de estos archivos, el navegador debía realizar varias peticiones a un servidor Web para bajar todos los archivos que conforman a un applet. Sin embargo, al combinar dichos archivos en uno solo mediante los archivos JAR, la petición se reduce a una sola y además la transferencia es más eficiente ya que se vuelve más rápida debido a la compresión.

Un archivo JAR consiste en un archivo sencillo que contiene una colección de archivos ZIP.

La herramienta jar se encuentra en el JDK para comprimir los archivos de nuestra elección. La sintaxis es la siguiente:

jar [opciones] destino [opciones] archivos_de_entrada.

Las opciones son las siguientes:

- c: Crea un nuevo o archivo vacío.
- t: Lista la tabla de contenidos.
- x : Extrae todos los archivos.
- x archivo: Extrae el archivo.
- f : Da el nombre al archivo.
- m: El primer argumento será el nombre del archivo.
- v: Genera una descripción de lo que hace el archivo jar.

Aquí hay varios ejemplos de cómo se usa esta herramienta:

jar cf myJarFile.jar *.class

Esto crea un archivo JAR llamado **myJarFile.jar** que contiene todas las clases del directorio actual.

jar cmf myJarFile.jar myManifestFile.mf *.class

Esto crea lo mismo que el anterior ejemplo pero con un archivo extra llamado **myManifetFile.mf.**

jar tf myJarFile.jar

Produce una tabla de contenido de los archivos dentro de **myJarFile.jar**

jar tvf myJarFile.jar

Da información más detallada acerca de los archivos contenidos dentro del archivo jar.

jar cvf myApp.jar audio classes image

Suponiendo que **audio**, **classes** e **images** son subdirectorios, este comando combina todos éstos dentro de **myApp.jar.** La opción "***v***" nos indica para saber si funcionó nuestro comando.

Si creamos un archivo JAR usando la opción "***0***", dicho archivo puede estar contenido dentro del **CLASSPATH**:

CLASSPATH="lib1.jar;lib2.jar;"

Entonces Java utilizará dichos archivos para buscar las clases contenidas en ellos.

La herramienta jar también tiene desventajas comparada con la herramienta ZIP, por ejemplo, no podemos agregar archivos a algún archivo JAR existente, ni eliminarlos, ni cambiarlos. Sin embargo, su gran ventaja es precisamente que son multiplataforma por lo cual serán reconocidos por cualquier sistema operativo con Java.

Serialización de Objetos.

El concepto de "***serialización***" nos permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bytes que pueden ser regenerados en el objeto original. Esto es particularmente útil para trabajo en red, porque podemos transferir un objeto en un sistema operativo X y reconstruirlo sin ningún problema en otro sistema operativo Y.

La ***serialización*** fue agregada al lenguaje por dos razones principalmente. La primera es la **RMI** de Java ("**remote method invocation**") y la segunda es los famosos **Java Beans**.

En realidad la ***serialización*** es bastante simple una vez que implementamos la interfaz **Serializable**. El siguiente ejemplo prueba la ***serialización*** creando un "gusano" de objetos enlazados. Cada uno de ellos contiene el siguiente segmento del gusano así como un arreglo de referencias de una clase distinta, **Data**:

```
//Gusano.java
//Probando la serializacion
import java.io.*;
class Data implements Serializable{
        private int i;
        Data(int x){ i=x;}
        public String toString(){
```

```
                return Integer.toString(i);
        }
}

public class Gusano implements Serializable{
        private static int r(){
                return (int)(Math.random()*10);
                }
        private Data[] d={new Data( r() ), new Data( r()), new Data(r())};
        private Gusano next;
        private char c;
        Gusano(int i,char x){
                System.out.println("Contuctor de Gusano: "+i);
                c=x;
                if(--i>0)
                        next=new Gusano(i, (char)(x+1));
                }
        Gusano(){
                System.out.println("Constructor Default");
                }
        public String toString(){
                String s=":" + c + "(";
                for(int i=0;i<d.length;i++)
                        s+=d[i].toString();
                s+=")";
                if(next!=null)
                        s+=next.toString();
                return s;
                }
        public static void main(String[] args)throws
IOException,ClassNotFoundException{
                Gusano w=new Gusano(6,'a');
                System.out.println("w= "+w);
                ObjectOutputStream out=new ObjectOutputStream(
                                new FileOutputStream("gusano.out"));
                out.writeObject("Almecen de gusanos");
                out.writeObject(w);
                out.close();
                ObjectInputStream in=new ObjectInputStream(
                                new FileInputStream("gusano.out"));
                String s=(String)in.readObject();
                Gusano w2=(Gusano)in.readObject();
                System.out.println(s+", w2 = "+w2);
                ByteArrayOutputStream bout=
                        new ByteArrayOutputStream();
                ObjectOutputStream out2=new ObjectOutputStream(bout);
                out2.writeObject("Almacen de gusanos");
```

```
            out2.writeObject(w);
            out2.flush();
            ObjectInputStream in2=new ObjectInputStream(
                        new ByteArrayInputStream(
                                    bout.toByteArray()));
            s=(String)in2.readObject();
            Gusano w3=(Gusano)in2.readObject();
            System.out.println(s+",w3= "+w3);
            }
}
```

En sí la ***serialización*** resulta muy simple, una vez que **ObjectOutputStream** es creado, **writeObject**() "***serializa***" el objeto. La primera sección del código lee y escribe a un archivo y la segunda escribe y lee un objeto **ByteArray**. La salida de la corrida fue:

```
Gusano constructor: 6
Gusano constructor: 5
Gusano constructor: 4
Gusano constructor: 3
Gusano constructor: 2
Gusano constructor: 1
w= :a(121):b(893):c(295):d(025):e(775):f(085)
Almecen de gusanos, w2 = :a(121):b(893):c(295):d(025):e(775):f(085)
Almacen de gusanos,w3= :a(121):b(893):c(295):d(025):e(775):f(085)
```

Veamos otro ejemplo donde enviamos un archivo para saber si a través de él podemos recuperar el objeto original.

Controlando la serialización.

También existen casos donde necesitamos un caso especial de ***serialización*** en donde no queramos ***serializar*** porciones del objeto. Podemos controlar el proceso de ***serialización*** gracias a la palabra reservada "**transient**", la cual nos permite desactivar la ***serialización*** campo por campo. *

***aunque también existe la interfaz Externalizable como segunda opción.**

Supongamos que tenemos un objeto con un campo Login que contiene información de alguna sesión en particular. Sopongamos que una vez almacenado el login, almacenamos sus datos pero no almacenamos la contraseña. La manera más sencilla de lograrlo es haciendo el campo de contaseña como transient.

```
//Logon.java
//Uso de transient
import java.io.*;
```

```
import java.util.*;
class Logon implements Serializable{
        private Date fecha=new Date();
        private String username;
        private transient String password;
        Logon(String name,String pwd){
                username=name;
                password=pwd;
                }
        public String toString(){
                String pwd=(password==null)?"(n/a)":password;
                return "logon info:\n "+"username: "+username+
                        "\n date: "+fecha+"\n password:"+pwd;
                }
        public static void main(String[] args)throws IOException,
ClassNotFoundException{
                Logon a=new Logon("Spiderman","PeterParker");
                System.out.println("logon a= "+a);
                ObjectOutputStream o=new ObjectOutputStream(
                        new FileOutputStream("Logon.out"));
                o.writeObject(a);
                o.close();
                int segundos=5;
                long t=System.currentTimeMillis()+segundos*1000;
                while(System.currentTimeMillis()<t)
                        ;
                ObjectInputStream in=new ObjectInputStream(
                                new FileInputStream("Logon.out"));
                System.out.println("Recuperando objeto en "+new Date());
                a=(Logon)in.readObject();
                System.out.println("logon a= "+a);
                }
        }
```

Tokenizing.

Tokenizing es el proceso de romper una secuencia de caracteres en una secuencia de "***tokens***". Por ejemplo, los ***tokens*** pueden ser palabras, las cuales están delimitadas por espacios en blanco y puntación. Existen dos clases que nos provee Java para el proceso de ***tokenizing***: **StreamTokenizer** y **StringTokenizer**.

StreamTokenizer

Esta clase trabaja solamente con objetos InputStream, por lo cual pertenece a la librería I/O. Consideremos el siguiente programa que cuenta la ocurrencia de una palabra en un archivo:

```
//WordCount.java
import java.io.*;
import java.util.*;
class Counter{
        private int i=1;
        int read(){ return i;}
        void increment(){i++;}
        }
public class WordCount{
        private FileReader file;
        private StreamTokenizer st;
        private TreeMap counts=new TreeMap();
        WordCount(String filename)throws FileNotFoundException{
                try{
                        file=new FileReader(filename);
                        st=new StreamTokenizer(
                                new BufferedReader(file));
                        st.ordinaryChar('.');
                        st.ordinaryChar('-');
                        }catch(FileNotFoundException f){
                           System.err.println("No se encontro el archivo:
                                        "+filename);

                            throw f;
                        }
                }
        void cleanup(){
                try{
                        file.close();
                        }catch(IOException e){
                                System.err.println(
                                "file.close() sin éxito");
                        }
                }
        void countWords(){
                try{
                while(st.nextToken()!=StreamTokenizer.TT_EOF){
                                String s;
                                switch(st.ttype){
                                        case StreamTokenizer.TT_EOL:
                                                s=new String("EOL");
```

```
                                        break;
                                    case StreamTokenizer.TT_NUMBER:
                                    s=Double.toString(st.nval);
                                        break;
                                    case StreamTokenizer.TT_WORD:
                                        s=st.sval;
                                        break;
                                    default:

s=String.valueOf((char)st.ttype);
                                    }
                                if(counts.containsKey(s))
                                    ((Counter)counts.get(s)).increment();
                                else
                                    counts.put(s,new Counter());
                                }
                            }catch(IOException e){
                                    System.err.println(
                                        "st.nextToken() sin exito");
                                }
                    }
                Collection values(){
                    return counts.values();
                    }
                Set keySet(){return counts.keySet();}
                Counter getCounter(String s){
                    return (Counter)counts.get(s);
                    }
                public static void main(String[] args)throws FileNotFoundException{
                    WordCount wc=new WordCount(args[0]);
                    wc.countWords();
                    Iterator keys=wc.keySet().iterator();
                    while(keys.hasNext()){
                        String key=(String)keys.next();
                        System.out.println(key+":  "
                            +wc.getCounter(key).read());
                        }
                    wc.cleanup();
                    }
                }
```

Las palabras se presentan en orden gracias al objeto **TreeMap** que automáticamente organiza sus claves en orden.

Para abrir el archivo, utilizamos **FileReader** y lo convertimos en "*palabras*" con ayuda de un **BufferedReader** dentro del objeto **StreamTokenizer**. En el objeto **StreamTokenizer**

existe una lista predeterminada de separadores, a los cuales podemos agregar los nuestros. Por ejemplo, **ordinaryChar**() indica que dicho carácter no nos interesa, por lo cual no será incluido en la lista resultante. Los ***tokens*** resultantes pueden ser EOF, números, cadenas o un carácter simple. La variable **ttype** nos indica el tipo de ***token***.

Una vez que encontramos un ***token***, el objeto **counts** de **TreeMap** nos indica si dicho **token** ya existe, en caso afirmativo, se aumenta el contador respectivo.

El método **cleanup**() es también importante porque realiza el cierre del archivo.

StringTokenizer.

Aunque no forma parte de la librería I/O, la clase **StringTokenizer** tiene una funcionalidad muy parecida a la clase **StreamTokenizer**.

Un **StringTokenizer** regresa los ***tokens*** dentro de una cadena . Estos ***tokens*** son caracteres consecutivos delimitados por tabuladores, espacios y nuevas líneas. Así, los ***tokens*** de la cadena *"Donde está Iskra?"* son *"Donde"*, *"está"*, *"Iskra*?". Como en **StreamTokenizer**, podemos indicarle a **StringTokenizer** romper la entrada en cualquier lugar que queramos, pero esto lo hacemos en el constructor como segundo argumento de éste, el cual es una cadena **String** de delimitadores.

Para obtener los **tokens** dentro de **StringTokenizer** utilizamos el método **nextToken**(), en caso de que no existan más ***tokens*** regresa el valor **null**.

Veamos un ejemplo:

```
//AnalizaSentencia.java
import java.util.*;
public class AnalizaSentencia{
        public static void main(String[] args){
                analiza("Yo estoy feliz de hacer esto!");
                analiza("Yo estoy triste de hacer esto");
                analiza("No! Yo estoy feliz");
                analiza("Yo estoy triste de hacer esto");
                analiza("No! Yo estoy triste");
                analiza("Estas triste de hacer esto?");
                analiza("Estas feliz de hacer esto?");
                analiza("Fuiste tu! Yo estoy feliz");
                analiza("Fuiste tu! Yo estoy triste");
                }
        static StringTokenizer st;
        static void analiza(String s){
                prt("\nNueva frase >>" +s);
                boolean triste=false;
```

```
st=new StringTokenizer(s);
while(st.hasMoreTokens()){
        String token=next();
        if(!token.equals("Yo")&&!token.equals("Estas"))
                continue;
        if(token.equals("Yo")){
                String tk2=next();
                if(!tk2.equals("estoy"))
                        break;
                else{
                        String tk3=next();
                        if(tk3.equals("triste")){
                                triste=true;
                                break;
                                }
                        if(tk3.equals("feliz"))
                                break;
                        }
                }
        if(token.equals("Estas")){
                String tk2=next();
                if(tk2.equals("triste")){
                        triste=true;
                        break;
                        }
                if(tk2.equals("feliz"))
                        break;
                }
        }
if(triste) prt("triste detectado");
}
static String next(){
        if(st.hasMoreTokens()){
                String s=st.nextToken();
                prt(s);
                return s;
                }
        else
                return "";
        }
static void prt(String s){
        System.out.println(s);
        }
}
```

Ejemplo de aplicación: Verificando el estilo de Java.

Como sabemos, Java tiene un estilo propio para el nombre de variables, métodos, clases, etc. El siguiente ejemplo verifica que nuestras clases cumplan con dicho estilo. El ejemplo abre el directorio actual y extrae todas lor archivos .java.

```
//ClassScanner.java
import java.io.*;
import java.util.*;
class MultiStringMap extends HashMap{
        public void add(String key, String value){
                if(!containsKey(key))
                        put(key,new ArrayList());
                ((ArrayList)get(key)).add(value);
                }
        public ArrayList getArrayList(String key){
                if(!containsKey(key)){
                        System.err.println("ERROR: no encuentro key: "
                                        +key);

                        System.exit(1);
                        }
                return (ArrayList)get(key);
                }
        public void printValues(PrintStream p){
                Iterator k=keySet().iterator();
                while(k.hasNext()){
                        String oneKey=(String)k.next();
                        ArrayList val=getArrayList(oneKey);
                        for(int i=0;i<val.size();i++)
                                p.println((String)val.get(i));
                        }
                }
        }
        public class ClassScanner{
                private File path;
                private String[] fileList;
                private Properties classes=new Properties();
                private MultiStringMap
                        classMap=new MultiStringMap(),
                        identMap=new MultiStringMap();
                private StreamTokenizer in;
                public ClassScanner() throws IOException{
                        path=new File(".");
                        fileList=path.list(new JavaFilter());
                        for(int i=0;i<fileList.length;i++){
```

```
                System.out.println(fileList[i]);
                try{
                        scanListing(fileList[i]);
                        }catch(FileNotFoundException f){
                                System.err.println(
                                "No se encontro "+fileList[i]);
                        }
                }
        }
void scanListing(String fname)throws IOException{
        in=new StreamTokenizer(new BufferedReader(
                        new FileReader(fname)));
        in.ordinaryChar('/');
        in.ordinaryChar('.');
        in.wordChars('_','_');
        in.eolIsSignificant(true);
        while(in.nextToken()!=StreamTokenizer.TT_EOF){
                if(in.ttype==StreamTokenizer.TT_WORD){
                        if(in.sval.equals("class") ||
                        in.sval.equals("interface")){
                                while(in.nextToken()!=
                                StreamTokenizer.TT_EOF &&
                        in.ttype!=StreamTokenizer.TT_WORD)
                                        ;
                                classes.put(in.sval,in.sval);
                                classMap.add(fname,in.sval);
                                }
                if(in.sval.equals("import")||
                        in.sval.equals("package"))
                                discardLine();
                else
                        identMap.add(fname,in.sval);
                        }
                }
        }
void discardLine() throws IOException{
        while(in.nextToken()!=StreamTokenizer.TT_EOF
                && in.ttype!=StreamTokenizer.TT_EOL)
                        ;
        }
void eatComments() throws IOException{
        if(in.nextToken()!=StreamTokenizer.TT_EOF){
                if(in.ttype=='/')
                        discardLine();
                else if(in.ttype!='*')
                        in.pushBack();
                else
```

```
                        while(true){
                            if(in.nextToken()==
                            StreamTokenizer.TT_EOF)
                                break;
                            if(in.ttype=='*')
                                if(in.nextToken()!=
                            StreamTokenizer.TT_EOF&&
                            in.ttype=='/')
                                break;
                            }
                    }
                }
            public String[] classNames(){
                String[] result=new String[classes.size()];
                Iterator e=classes.keySet().iterator();
                int i=0;
                while(e.hasNext())
                    result[i++]=(String)e.next();
                return result;
                }
            public void checkClassNames(){
                Iterator files=classMap.keySet().iterator();
                while(files.hasNext()){
                    String file=(String)files.next();
                    ArrayList cls=classMap.getArrayList(file);
                    for(int i=0;i<cls.size();i++){
                        String className=(String)cls.get(i);
                        if(Character.isLowerCase(
                            className.charAt(0)))
                            System.out.println(
"Error de estilo en archivo: "+file+",class "+className);
                        }
                    }
                }
            public void checkIdentNames(){
                Iterator files=identMap.keySet().iterator();
                ArrayList reportSet=new ArrayList();
                while(files.hasNext()){
                    String file=(String)files.next();
                    ArrayList ids=identMap.getArrayList(file);
                    for(int i=0;i<ids.size();i++){
                        String id=(String)ids.get(i);
                        if(!classes.contains(id)){
                            if(id.length()>=3&&
                            id.equals(id.toUpperCase()))
                                continue;
                            if(Character.isUpperCase(
```

```
                                        id.charAt(0))){
                                        if(reportSet.indexOf(file+id)
                                            ==-1){
                                            reportSet.add(file+id);
                                            System.out.println(
                                    "Error de Estilo en ident: "
                                            +file+", ident: "+id);
                                            }
                                    }
                                }
                            }
                        }
                }
        static final String usage=
                "Uso: \n"+
                "ClassScanner classnames –a\n"+
                "\tAgega todas los nombres de clases en este \n"+
                "\tdirectorio denrto del archivo\n"+
                "\tllamado 'classnames'\n"+
                "\tCheca que todos los archivo java de este directorio\n"+
                "\tusando el archivo 'classname'";
        private static void usage(){
                System.err.println(usage);
                System.exit(1);
                }
        public static void main(String[] args)throws IOException{
                if(args.length<1 || args.length>2)
                        usage();
                ClassScanner c=new ClassScanner();
                File old=new File(args[0]);
                if(old.exists()){
                        try{
                                InputStream oldList=
                                        new BufferedInputStream(
                                                new FileInputStream(old));
                                c.classes.load(oldList);
                                oldList.close();
                                }catch(IOException e){
                                        System.err.println(
                                        "No se pudo abrir :"+old+
                                        "para lectura");
                                        System.exit(1);
                                }
                        }
                if(args.length==1){
                        c.checkClassNames();
                        c.checkIdentNames();
```

```
        }
if(args.length==2){
        if(!args[1].equals("-a"))
                usage();
        try{
                BufferedOutputStream out=
                new BufferedOutputStream(
                        new FileOutputStream(args[0]));
        c.classes.store(out,"No se encontraron las clases");
                out.close();
                }catch(IOException e){
                        System.err.println("No se pudo escribir"
                                +args[0]);
                        System.exit(1);
                        }
                }
        }
}
class JavaFilter implements FilenameFilter{
        public boolean accept(File dir,String name){
                String f=new File(name).getName();
                return f.trim().endsWith(".java");
                }
        }
```

3 Creando Applets y GUI con Swing.

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.

Un applet básico.

Uno de los principales objetivos de la creación de Java es crear **applets**. Los **applets** son pequeños programas que corren dentro de un navegador de Red. Son herramientas poderosas que soportan la programación cliente-servidor, una de las ventajas de la Red.

Restricciones de un applet.

Programar dentro de un applet tiene sus limitaciones, ya que tenemos ciertas restricciones. El Java run-time security system vigila que cumplamos con los siguientes puntos:

1. ***Un applet no puede tocar el disco duro local***. Esto implica la lectura y escritura, ya que podríamos transmitir información confidencial de manera ilegal a través de Internet.
2. ***Un applet no puede ejecutar comandos del Sistema Operativo***.
3. ***Un applet debe conectarse vía socket***.
4. ***Un applet debería estar contenido dentro de una archivo JAR***. Dicho archivo debe combinar las clases de dicho applet así como imágenes y sonido que pueda contener.

Ventajas de un applet.

Los **applets** cuentan definitivamente con ventajas, especialmente al construir aplicaciones cliente/servidor o cualquier aplicación que involucre una red.

1. ***No existe instalación***. Un applet funciona verdaderamente como una aplicación independiente de la plataforma.
2. ***Actualización automática***. Cuando se actualiza un applet se realiza del lado del servidor, por lo que el cliente no tiene que preocuparse por esto, evitando actualizaciones de software cliente que generalmente son pesadas y caras.
3. ***No existe riesgo de daños al sistema***. Debido a las restricciones enunciadas anteriormente.

Un applet básico.

Los **applets** trabajan dentro de "*marcos de trabajo*" (***framework***). Heredan de la clase **JApplet** y sobrescriben sus métodos. Los métodos que controlan la creación y ejecución de un applet en una página Web son:

Método	Operación
init()	Llamado automáticamente cada vez que se inicializa un applet.
start()	Llamada después de **init**() y cada vez que el foco del navegador se encuentra en el applet.
stop()	Llamada cada vez que el navegador quita el foco del applet o termina sus operaciones.
destroy()	Llamada cada vez que el applet terminó de utilizarse.

Veamos un ejemplo de un applet sencillo:

```
//Applet1.java
import javax.swing.*;
import java.awt.*;
public class Applet1 extends JApplet{
        public void init(){
                getContentPane().add(new JLabel("Hola Applet!"));
                }
        }
```

Un **applet** no requiere método **main**(). En este programa, la única actividad es poner un campo de texto en pantalla. Como vemos, el applet hereda la clase **JApplet** y sobrescribe el

método **init**(), el cual es el responsable de poner cualquier componente en pantalla con el método **add**(). En el ejemplo, ponemos un "panel" (**getContentPane**()) dentro del cual ponemos un objeto **JLabel**.

Corriendo Applets en un navegador Web.

Para correr este programa, debemos ponerlo dentro de una página Web. Para poner un applet dentro de una página utilizamos el tag:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Existen diversas convenciones extras para el manejo de applets dentro de HTML.

Utilizando Appletviewer.

La JDK de Sun tiene una herramienta llamada Appletviewer que toma los tags <applet> de un archivo HTML y corre el applet independientemente del demás código HTML. Por lo que los ejemplos restantes de esta sección endrán comentados las siguientes líneas:

```
//<applet code=MiApplet width=200 height=100>
//</applet>
```

Así podemos utilizar la herramienta appletviewer sin crear un HTML correspondiente para cada applet creado.

```
//Applet1b.java
//<applet code=Applet1b width=200 height=100>
//</applet>
import javax.swing.*;
import java.awt.*;
public class Applet1b extends JApplet{
        public void init(){
                getContentPane().add(new JLabel("Hola Applet!"));
                }
        }
```

Ahora, utilizando el comando después de haber compilado el código anterior,

appletviewer Applet1b.java

podemos ver el applet.

Corriendo applets desde la línea de comandos.

Existen varias aplicaciones donde realizamos aplicaciones gráficas demasiado pesadas o totalmente independientes las páginas Web. Por diversas razones son importantes las aplicaciones gráficas, por ejemplo, los ambientes Macintosh no tienen líneas de comandos.

Para crear un applet que también pueda ejecutarse independientemente de un navegador Web, simplemente agregaremos el método **main**() donde crearemos un objeto de la clase **JFrame**.

```
//Applet1c.java
//<applet code=Applet1c width=200 height=50>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Applet1c extends JApplet{
        public void init(){
                getContentPane().add(new JLabel("Hola Applet!"));
                }
        public static void main(String[] args){
                JApplet applet=new Applet1c();
                JFrame frame=new JFrame("Applet1c");
                Consola.setupClosing(frame);
                frame.getContentPane().add(applet);
                frame.setSize(100,50);
                applet.init();
                applet.start();
                frame.setVisible(true);
                }
        }
```

Podemos ver que en el método main() el applet es inicializado explícitamente ya que el navegador no lo hace por nosotros. También es importante la línea :

frame.setVisible(true);

ya que despliega lo que se ve en pantalla. El código de **Consola** es el de una utilería dentro del paquete **com.serge**.**swing**. Esta clase contiene métodos para el título, crear el applet y cerrarlo.

```
//Consola.java
package com.serge.swing;
import javax.swing.*;
import java.awt.event.*;
public class Consola{
        public static String title(Object o){
                String t=o.getClass().toString();
                if(t.indexOf("class")!= -1)
                        t=t.substring(6);
                return t;
                }
        public static void setupClosing(JFrame frame){
                frame.addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e){
                                System.exit(0);
                                }
                        });
                }
        public static void run(JFrame frame,int width,int height){
                setupClosing(frame);
                frame.setSize(width,height);
                frame.setVisible(true);
                }
        public static void run(JApplet applet,int width,int height){
                JFrame frame=new JFrame(title(applet));
                setupClosing(frame);
                frame.getContentPane().add(applet);
                frame.setSize(width,height);
                applet.init();
                applet.start();
                frame.setVisible(true);
                }
        public static void run(JPanel panel,int width,int height){
                JFrame frame=new JFrame(title(panel));
                setupClosing(frame);
                frame.getContentPane().add(panel);
                frame.setSize(width,height);
                frame.setVisible(true);
                }
        }
```

El método **setupClosing**() es utilizado para cerrar el **Frame**.
El método **run**() está sobrecargado para trabajar con **JApplet**, **JPanel** y **JFrame**.

Entonces podemos crear cualquier applet desde **main**() con la siguiente línea:

Consola.run(new MyClase(),500,300);

Veamos el ejemplo anterior modificado para usar Consola:

```
//Applet1d.java
//<applet code=Applet1d width=100 height=50>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Applet1d extends JApplet{
        public void init(){
                getContentPane().add(new JLabel("Hola Applet!"));
                }
        public static void main(String[] args){
                Consola.run(newApplet1d(),100,50);
                }
        }
```

Creando un botón.

Hacer un botón es bastante sencillo: simplemente llamamos al constructor de JButton con la etiqueta que queremos que tenga el botón.

```
//Boton1.java
//<applet code=Boton1 width=200 height=50>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Boton1 extends JApplet{
        JButton b1=new JButton("Boton1"),
                  b2=new JButton("Boton2");
        public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(b1);
                cp.add(b2);
                }
        public static void main(String[] args){
```

```
            Consola.run(new Boton1(),200,50);
            }
}
```

Capturando un evento.

Al compilar y ejecutar el ejemplo anterior notaremos que al hacer clic en los botones, no tienen funcionalidad alguna. Es aquí donde debemos escribir el código necesario para dicha funcionalidad.

La base del manejo de eventos dentro de **Swing** es escribir separar la interfaz (los componentes gráficos) de la implementación (el código que escribimos cuando ocurre determinado evento). Esto es una manera muy elegante de manejar la programación de eventos, y una vez que entendemos los conceptos básicos de los componentes **Swing**, el manejo de eventos es bastante simple.

Primero nos concentraremos en los eventos principales que generan los componentes utilizados. En el caso de **JButton**, el "***evento de interés***" es que el botón sea presionado. Para registrar nuestro interés en dicho evento, llamamos al método **addActionListener**() de **JButton**. Este método recibe como argumento un objeto que implemente la interfaz **ActionListener**.

¿Y cuál es el resultado de que el botón sea presionado? Sería conveniente ver algún cambio en la pantalla, entonces crearemos un nuevo componente de **Swing** llamado **JTextField**. Este es un campo de texto donde el usuario puede escribir. Una vez que el objeto **JTextField** existe, podemos modificar su contenido usando el método **setText**().

```
//Boton2.java
//<applet code=Boton2 width=200 height=75>
//</applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.serge.swing.*;
public class Boton2 extends JApplet{
        JButton b1=new JButton("Boton 1"),
                b2=new JButton("Boton2");
        JTextField txt=new JTextField(10);
        class BL implements ActionListener{
                public void actionPerformed(ActionEvent e){
                        String nombre=
                                ((JButton)e.getSource()).getText();
                        txt.setText(nombre);
                        }
```

```
        }
BL a1=new BL();
public void init(){
        b1.addActionListener(a1);
        b2.addActionListener(a1);
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(txt);
        }
public static void main(String[] args){
        Consola.run(new Boton2(),200,75);
        }
}
```

Como vemos, agregar cualquier componente **Swing** toma los mismos pasos, así es con todos sus componentes. El argumento de **actionPerformed**() es un objeto **ActionEvent** que contiene la información de quién produjo el evento. El método **getSource**() produce el objeto que originó el evento, en este caso es un objeto **JButton**. El método **getText**() regresa el texto que tiene el botón y lo ponemos en el panel de texto **txt** con el método **setText**().

Es conveniente implementar al código de **ActionListener** como una clase anónima interna, ya que por lo general utilizamos una sola instancia a dicha clase. Veamos el código de **Boton2.java** con una clase interna:

```
//Boton2b.java
//<applet code=Boton2b width=200 height=75>
//</applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.serge.swing.*;
public class Boton2b extends JApplet{
        JButton b1=new JButton("Boton 1"),
                b2=new JButton("Boton2");
        JTextField txt=new JTextField(10);
        ActionListener a1=new ActionListener(){
                public void actionPerformed(ActionEvent e){
                        String nombre=
                                ((JButton)e.getSource()).getText();
                        txt.setText(nombre);
                        }
                };
        public void init(){
                b1.addActionListener(a1);
```

```
        b2.addActionListener(a1);
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(txt);
        }
    public static void main(String[] args){
        Consola.run(new Boton2b(),200,75);
        }
    }
```

Áreas de Texto.

Un objeto JTextArea es muy similar a un JTextField. La diferencia es que puede tener múltiples líneas de entrada por lo que tiene más funcionalidad. Un método muy útil es append() con el cual podemos editar la salida dentro de JTextArea. Veamos un ejemplo:

```
//TextArea.java
//<applet code=TextArea width=475 height=425>
//</applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class TextArea extends JApplet{
    JButton b=new JButton("Agregar datos!"),
            c=new JButton("Borrar datos!");
    JTextArea t=new JTextArea(20,40);
    public void init(){
    final String llenar="Este es simple texto para llenar el area"+
                        "\ntexto, lo interesante es ver que un "+
                        "\nobjeto JTextArea es capaz de almacenar"+
                        "\nvarias lineas y no solo una. Bla Bla bla..."+
                        "\nbla.....bla.....bla.....bla..";
        b.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                t.setText(llenar);
                }
            });
        c.addActionListener(new ActionListener (){
            public void actionPerformed(ActionEvent e){
                t.setText("");
                }
            });
```

```
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JScrollPane(t));
        cp.add(b);
        cp.add( c );
        }
public static void main(String[] args){
        Consola.run(new TextArea(),475,425);
        }
}
```

De nuevo podemos observar que los componentes son bastante sencillos y directos. Lo nuevo aquí es el uso de un objeto **JScrollPanel**, el cual representa una barra de desplazamiento cuando existe mucho texto en pantalla.

Layout managers.

La manera en que los componentes son distribuidos en la pantalla es diferente en Java a cualquier sistema gráfico. Primero, todo es código: no hay "elementos" que controlen la posición de los componentes. Luego, la manera en que son distribuidos los elementos es por un "layout manager" que decide cómo se van a acomodar los elementos al utilizar el método add().

Entre cada layout manager existen diferencias en cómo distribuyen los elementos. Las clases **JApplet**, **JFrame**, **JWindow** y **JDialog** producen un **Container** con el método **getContentPane**() que contiene a todos los elementos. El objeto **Container** tiene un método llamado **setLayout**() que permite escoger el layout manager.

BorderLayout.

El layout predeterminado es precisamente el **BorderLayout**. Este layout toma cualquier elemento que agreguemos y lo coloca en el centro, desplazando los componentes existentes a los lados.

Sin embargo, este layout nos permite otro tipo de control ya que cuenta con cuatro regiones y un área central. Cuando agregamos algo al panel, podemos utilizar el método **add**() sobrecargado que toma una constante como primer argumento. Este puede ser cualquier de los siguientes:

BorderLayout.NORTH
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.CENTER

Si no se especifica el lugar donde se colocará el objeto, **CENTER** será el predeterminado.

```
//BorderLayout1.java
//<applet code=BorderLayout1 width=300 height=250>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class BorderLayout1 extends JApplet{
        public void init(){
                Container cp=getContentPane();
                cp.add(BorderLayout.NORTH,new Jbutton("Norte"));
                cp.add(BorderLayout.SOUTH,new Jbutton("Sur"));
                cp.add(BorderLayout.EAST,new Jbutton("Este"));
                cp.add(BorderLayout.WEST,new Jbutton("Oeste"));
                cp.add(BorderLayout.CENTER,new Jbutton("Centro"));
                }
        public static void main(String[] args){
                Consola.run(new BorderLayout1(),300,250);
                }
        }
```

FlowLayout.

Este layout simplemente distribuye todos los componentes en forma horizontal de izquierda a derecha hasta que se acabe el espacio, entonces crea un nuevo renglón y continua con el mismo proceso.

Veamos un ejemplo:

```
//FlowLayout1.java
//<applet code=FlowLayout1 width=300 height=250>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class FlowLayout1 extends JApplet{
        public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                for(int i=0;i<20;i++)
                        cp.add(new Jbutton("Boton"+i));
                }
        public static void main(String[] args){
                Consola.run(new FlowLayout1(),300,200);
```

```
        }
}
```

GridLayout.

GridLayout nos permite distribuir los components como en una rejilla. En el constructor especificamos el número de renglones y columnas que necesitamos:

```
//GridLayout1.java
//<applet code=GridLayout1 width=300 height=250>
//</applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class GridLayout1 extends JApplet{
    public void init(){
        Container cp=getContentPane();
        cp.setLayout(new GridLayout(7,3));
        for(int i=0;i<20;i++)
            cp.add(new JButton("Boton"+i));
        }
    public static void main(String[] args){
        Consola.run(new GridLayout1(),300,250);
        }
}
```

GridBigLayout.

Este layout es el que nos proporciona mayor control sobre la posición de nuestros elementos. Sin embargo, también es el más complicado y bastante difícil de entender. Existen libros enteros dedicados a este layout manager por lo que se recomienda su lectura si se interesa profundizar en el tema.

BoxLayout.

Este layout nos brinda muchos de los beneficios de GridBigLayout sin su complejidad, por lo que lo podemos utilizar más fácilmente. BoxLayout permite el control de la posición de los elementos ya sea vertical u horizontalmente además de especificar el espacio entre ellos.

```
//BoxLayout1.java
//<applet code=BoxLayout1 width=450 height=200></applet>
import javax.swing.*;
```

```
import java.awt.*;
import com.serge.swing.*;
public class BoxLayout1 extends JApplet{
        public void init(){
                JPanel jpv=new JPanel();
                jpv.setLayout(
new BoxLayout(jpv,BoxLayout.Y_AXIS));
                for(int i=0;i<5;i++)
                        jpv.add(new JButton(“”+i));
                JPanel jph=new JPanel();
                jph.setLayout(
new BoxLayout(jph,BoxLayout.X_AXIS));
                for(int i=0;i<5;i++)
                        jph.add(new JButton(“”+i));
                Container cp=getContentPane();
                cp.add(BorderLayout.EAST,jpv);
                cp.add(BorderLayout.SOUTH,jph);
                }
        public static void main(String[] args){
                Consola.run(new BoxLayout1(),450,200);
                }
        }
```

El constructor de BoxLayout es un poco diferente a los vistos anteriormente. Como argumentos recibe el objeto Container y la diección como segundo argumento.

Para simplificar, existe un contenedor especial llamado Box que utiliza un BoxLayout como su método nativo. El siguiente ejemplo utiliza Box para colocar los elementos vertical y horizontalmente:

```
//Box1.java
//<applet code=Box1 width=450 height=200></applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Box1 extends JApplet{
        public void init(){
                Box bv=Box.createVerticalBox();
                for(int i=0;i<5;i++)
                        bv.add(new JButton(“”+i));
                Box bh=Box.createHorizontalBox();
                for(int i=0;i<5;i++)
                        bh.add(new JButton(“”+i));
                Container cp=getContentPane();
                cp.add(BorderLayout.EAST,bv);
                cp.add(BorderLayout.SOUTH,bh);
```

```
        }
    public static void main(String[] args){
        Consola.run(new Box1(),450,200);
        }
    }
```

También podemos "pegar" elementos en blanco de tal manera que controlemos el espacio de los componentes.

```
//Box2.java
//<applet code=Bo2 width=450 height=300></applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Box2 extends JApplet{
    public void init(){
        Box bv=Box.createVerticalBox();
        for(int i=0;i<5;i++){
            bv.add(new JButton(""+i));
            bv.add(Box.createVerticalStrut(i*10));
            }
        Box bh=Box.createHorizontalBox();
        for(int i=0;i<5;i++){
            bh.add(new JButton(""+i));
            bh.add(Box.createHorizontalStrut(i*10));
            }
        Container cp=getContentPane();
        cp.add(BorderLayout.EAST,bv);
        cp.add(BorderLayout.SOUTH,bh);
        }
    public static void main(String[] args){
        Consola.run(new Box2(),450,300);
        }
    }
```

También podemos utilizar un componente llamado "glue" que tiene un propósito similar.

```
//Box3.java
//<applet code=Box3 width=450 height=300></applet>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Box3 extends JApplet{
    public void init(){
        Box bv=Box.createVerticalBox();
        bv.add(new JLabel("Hola"));
```

```
                bv.add(Box.createVerticalGlue());
                bv.add(new JLabel("Mundo"));
                bv.add(Box.createVerticalGlue());
                bv.add(new JLabel("Applet"));
                Box bh=Box.createHorizontalBox();
                bh.add(new JLabel("Hola"));
                bh.add(Box.createHorizontalGlue());
                bh.add(new JLabel("Mundo"));
                bh.add(Box.createHorizontalGlue());
                bh.add(new JLabel("Applet"));
                bv.add(Box.createVerticalGlue());
                bv.add(bh);
                bv.add(Box.createVerticalGlue());
                getContentPane().add(bv);
                }
        public static void main(String[] args){
                Consola.run(new Box3(),450,300);
                }
        }
```

Veamos otro ejemplo:

```
//Box4.java
//<applets code=Box4 width= 450 height=300></applets>
import javax.swing.*;
import java.awt.*;
import com.serge.swing.*;
public class Box4 extends JApplet{
        public void init(){
                Box bv=Box.createVerticalBox();
                bv.add(new JButton("Top"));
                bv.add(Box.createRigidArea(new Dimension(120,90)));
                bv.add(new JButton("Bottom"));
                Box bh=Box.createHorizontalBox();
                bh.add(new JButton("Left"));
                bh.add(Box.createRigidArea(new Dimension(160,180)));
                bh.add(new JButton("Right"));
                bv.add(bh);
                getContentPane().add(bv);
                }
        public static void main(String[] args){
                Consola.run(new Box4(),450,300);
                }
        }
```

Modelo de Eventos de Swing.

El modelo de eventos de **Swing** permite que un componente "*dispare*" un evento. Cada tipo de evento es representado por una clase distinta. Cuando un evento es disparado, éste es recibido por uno o más "**Listeners**", que actúan sobre el evento. Así, la fuente del evento y el lugar donde manejamos el evento pueden ser manejados por separado.

Cada **listener** de un evento es un objeto de una clase que implementa a una interfaz en particular. Entonces, el programador debe crear el objeto listener y registrarlo con el componente que disparó el evento. Este registro es manejado al llamar al método **addXXXListener**() en el componente que disparó el evento. Cuando se crea un listener, la única restricción es que debe implementar la interfaz apropiada.

Eventos y tipos de listeners.

Todos los componentes de **Swing** incluyen los métodos **addXXXListener**() y **removeXXXListener**(). La siguiente tabla incluye los eventos básicos asociados con los componentes que pueden generar dichos eventos.

Evento, listener y métodos	Componentes que soportan este evento.
ActionEvent **ActionListener** **addActionListener()** **removeActionListener()**	**JButton**, **JList**, **JTextField**, **JMenuItem** y sus clases derivadas **JCheckBoxMenuItem**, **JMenu** y **JpopupMenu**.
AdjustmentEvent **AdjustementListener** **addAdjustmentListener()** **removeAdjustmentListener()**	**JScrollbar**
ComponentEvent **ComponentListener** **addComponentListener()** **removeComponentListener()**	**JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollBar, JTextArea** y **JTextField**
ContainerEvent **ContainerListener** **addContainerListener()** **removeContainerListener()** **FocusEvent** **FocusListener** **addFocusListener()**	Container y clases derivadas como **JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog y JFrame.** **Component** y clases derivadas

removeFocusListener()	
KeyEvent **KeyListener** **addKeyListener()** **removeKeyListener()**	**Component** y sus clases derivadas.
MouseEvent **MouseListener** **addMouseListener()** **removeMouseListener()**	**Component** y sus clases derivadas.
MouseEvent **MouseMotionListener** **addMouseMotionListener()** **removeMouseMotionListener()**	**Component** y sus clases derivadas
WindowEvent **WindowListener** **addWindowListener()** **removeWindowListener()**	**Window** y clase derivadas como **JDialog, JFileDialog y JFrame**.
ItemEvent **ItemListener** **addItemListener()** **removeItemListener()**	**JCheckBox, JCheckBoxMenuItem, JComboBox y JList**
TextEventListener **TextListener** **addTextListener()** **removeTextListener()**	Todas las clases derivadas de **JTextComponent** incluyendo **JTextArea y JTextField**.

Una vez que sabemos cuáles eventos maneja un componente en particular, simplemente haremos:

1. **Tomar el nombre de la clase del evento y quitarle la palabra "Event". Agregamos la palabra "Listener". Esta será la interfaz que debemos implementar.**

2. **Implementar la interfaz anterior y escribir los métodos para los eventos que capturemos. Por ejemplo, si queremos hacer algo en particular cuando el ratón se mueva, entonces escribimos código para el método mouseMoved() de la interfaz MouseMotionListener.**

3. **Crear un objeto del listener del paso 2. Registrarlo dentro de nuestro componente con el método addXXXListener(). En este caso addMouseMotionListener().**

Aquí está la lista completa de interfaces:

Interfaz Listener	Métodos de la interfaz
ActionListener	**actionPerformed(ActionEvent)**
AdjustementListener	**adjustementValueChanged(AdjustementEvent)**
ComponentListener **ComponentAdapter**	**componentHidden(ComponentEvent)** **componentShown(ComponentEvent)** **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)**

Veamos un ejemplo:

```
//TrackEvent.java
//<applet code=TrackEvent width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.serge.swing.*;

public class TrackEvent extends JApplet{
        HashMap h=new HashMap();
        String[] event={
        "focusGained","focusLost","keyPressed","keyRelesead",
```

```
            "keyTyped","mouseClicked","mouseEntered",
            "mouseExited","mousePressed","mouseReleased",
            "mouseDragged","mouseMoved"};
    MyButton b1=new MyButton(Color.blue,"Test1"),
                b2=new MyButton(Color.red,"Test2");
    class MyButton extends JButton{
        void report(String field,String msg){
            ((JTextField)h.get(field)).setText(msg);
            }
        FocusListener f1=new FocusListener(){
            public void focusGained(FocusEvent e){
                report("focusGained",e.paramString());
                }
            public void focusLost(FocusEvent e){
                report("focusLost", e.paramString());
                }
            };
        KeyListener k1=new KeyListener(){
            public void keyPressed(KeyEvent e){
                report("keyPressed",e.paramString());
                }
            public void keyReleased(KeyEvent e){
                report("keyReleased",e.paramString());
                }
            public void keyTyped(KeyEvent e){
                report("keyTyped",e.paramString());
                }
            };
        MouseListener m1=new MouseListener(){
            public void    mouseClicked(MouseEvent e){
                report("mouseClicked",e.paramString());
                }
            public void mouseEntered(MouseEvent e){
                report("mouseEntered",e.paramString());
                }

            public void mouseExited(MouseEvent e){
                report("mouseExited",e.paramString());
                }
            public void mousePressed(MouseEvent e){
                report("mousePressed",e.paramString());
                }
            public void mouseReleased(MouseEvent e){
                report("mouseReleased",e.paramString());
                }
            };
        MouseMotionListener mm1=
```

```
            new MouseMotionListener(){
                public void mouseDragged(MouseEvent e){
                        report("mouseDraged",e.paramString());
                        }
                public void mouseMoved(MouseEvent e){
                        report("mouseMoved",e.paramString());
                        }
                };
        public MyButton(Color color, String label){
                super(label);
                setBackground(color);
                addFocusListener(f1);
                addKeyListener(k1);
                addMouseListener(m1);
                addMouseMotionListener(mm1);
                }
        }

public void init(){
        Container c=getContentPane();
        c.setLayout(new GridLayout(event.length+1,2));
        for(int i=0;i<event.length;i++){
                JTextField t=new JTextField();
                t.setEditable(false);
                c.add(new JLabel(event[i],JLabel.RIGHT));
                c.add(t);
                h.put(event[i],t);
                }
        c.add(b1);
        c.add(b2);
        }

public static void main(String[] args){
        Consola.run(new TrackEvent(),700,500);
        }
}
```

Catálogo de componentes Swing.

Botones

Swing tiene una gran variedad de botones. Todos los **botones**, **checkboxes**, **radio buttons** y los **ítems** son heredados de **AbstractButton**. El siguiente ejemplo muestra varios tipos de botones disponibles:

```
//Botones.java
//<applet code=Botones width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;
import com.serge.swing.*;

public class Botones extends JApplet{
        JButton jb=new JButton("JButton");
        BasicArrowButton up=new BasicArrowButton(
                                BasicArrowButton.NORTH),
                                down=new BasicArrowButton(
                                BasicArrowButton.SOUTH),
                                right=  new BasicArrowButton(
                                BasicArrowButton.EAST),
                                left=new BasicArrowButton(
                                BasicArrowButton.WEST);
        public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(jb);
                cp.add(new JToggleButton("JToggleButton"));
                cp.add(new JCheckBox("JCheckBox"));
                cp.add(new JRadioButton("JRadioButton"));
                JPanel jp=new JPanel();
                jp.setBorder(new TitledBorder("Direcciones"));
                jp.add(up);
                jp.add(down);
                jp.add(left);
                jp.add(right);
                cp.add(jp);
                }
        public static void main(String[] args){
                Consola.run(new Botones(),350,100);
                }
```

```
}
```

El objeto **BasicArrowButton** pertenece a la librería **javax.swing.plaf.basic**. El botón **JToggleButton** mantiene su estado una vez presionado.

Grupos de Botones.

Si queremos que los radio buttons se comporten de una manera particular, deben agregarse a un "grupo de botones".

```
//ButtonGroups.java
//<applet code=ButtonGroups width=500 height=300></applet>
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.awt.*;
import java.lang.reflect.*;
import com.serge.swing.*;
public class ButtonGroups extends JApplet{
        static String[] ids={ "June", "Ward", "Beaver",
                                "Wally","Eddie","Lumpy"};
        static JPanel makeBPanel(Class bClass,String[] ids){
                ButtonGroup bg=newButtonGroup();
                JPanel jp=new JPanel();
                String title=bClass.getName();
                title=title.substring(title.lastIndexOf('.')+1);
                jp.setBorder(new TitledBorder(title));
                for(int i=0;i<ids.length;i++){
                        AbstractButton ab=new JButton("Failed");
                        try{
                                Constructor ctor=bClass.getConstructor(
                                        new Class[]{String.class});
                                ab=(AbstractButton)ctor.newInstance(
                                        new Object[]{ids[i]});
                                }catch(Exception ex){
                                        System.err.println("no se pudo crear"
                                                +bClass);
                                }
                        bg.add(ab);
                        jp.add(ab);
                        }
                return jp;
                }
        public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
```

```
            cp.add(makeBPanel(JButton.class, ids));
            cp.add(makeBPanel(JToggleButton.class,ids));
            cp.add(makeBPanel(JCheckBox.class,ids));
            cp.add(makeBPanel(JRadioButton.class,ids));
            }
      public static void main(String[] args){
            Consola.run(new ButtonGroups(),500,300);
            }
      }
```

Icon

Se pueden utilizar los ***icons*** dentro de un objeto **JLabel** o cualquier cosa que herede de **AbstractButton**. Utilizar un ***icon*** dentro de **JLabel** es bastante directo. El siguiente ejemplo muestra las manera en que podemos utilizar **Icon** con botones.

```
//Iconos.java
//<applet code=Icons width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class Iconos extends JApplet{
      static String   path="c:/objetos/swing/images/";
      static Icon[] iconos={
            new ImageIcon(path+"icon0.gif"),
            new ImageIcon(path+"icon1.gif"),
            new ImageIcon(path+"icon2.gif"),
            new ImageIcon(path+"icon3.gif"),
            new ImageIcon(path+"icon4.gif"),
            };
      JButton jb=new JButton("JButton",iconos[3]),
            jb2=new JButton("Deshabilitar");
      boolean mad=false;
      public void init(){
            Container cp=getContentPane();
            cp.setLayout(new FlowLayout());
            jb.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent e){
                        if(mad){
                              jb.setIcon(iconos[3]);
                              mad=false;
                        }else{
                              jb.setIcon(iconos[0]);
                              mad=true;
```

```
                        }
                        jb.setVerticalAlignment(JButton.TOP);
                        jb.setHorizontalAlignment(JButton.LEFT);
                        }
                });
        jb.setRolloverEnabled(true);
        jb.setRolloverIcon(iconos[1]);
        jb.setPressedIcon(iconos[2]);
        jb.setDisabledIcon(iconos[4]);
        jb.setToolTipText("Yow!");
        cp.add(jb);
        jb2.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e){
                        if(jb.isEnabled()){
                                jb.setEnabled(false);
                                jb2.setText("Habilitado");
                                }else{
                                        jb.setEnabled(true);
                                        jb2.setText("Deshabilitado");
                                }
                        }
                });
        cp.add(jb2);
        }
public static void main(String[] args){
        Consola.run(new Iconos(),400,200);
        }
}
```

Text fields.

El siguiente ejemplo muestra el comportamiento del objeto **JTextField**:

```
//TextFields.java
//<applet code=TextField width=375 height=125></applets>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class TextFields extends JApplet{
        JButton b1=new JButton("Get Text"),
                b2=new JButton("Set Text");
        JTextField t1=new JTextField(30),
                   t2=new JTextField(30),
```

```
            t3=new JTextField(30);
String s=new String();
UpperCaseDocument ucd=new UppercaseDocument();
public void init(){
        t1.setDocument(ucd);
        ucd.addDocumentListener(new T1());
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        DocumentListener d1=new T1();
        t1.addActionListener(new T1A());
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(t1);
        cp.add(t2);
        cp.add(t3);
        }
class T1 implements DocumentListener{
        public void changeUpdate(DocumentEvent e){}
        public void insertUpdate(Docume ntEvent e){
                t2.setText(t 1.getText()) ;
                t3.setText( « Texto :  » +t1.getText() );
                }
        }
class T 1A implements ActionListener {
        private int count=0 ;
        public void actionPerformed(ActionEvent e){
                t3.setText("t1 Action Event "+count++);
                }
        }
class B1 implements ActionListener{
        public void actionPerformed(ActionEvent e){
                if(t1.getSelectedText()==null)
                        s=t1.getText();
                else
                        s=t1.getSelectedText();
                t1.setEditable(true);
                }
        }
class B2 implements Actio nListener{
        public void actionPerformed(ActionEvent e){
                ucd.setUpperCase(false);
                t1.setText("Insertado por el boton 2: "+s);
                ucd.setUpperCase(true);
                t1.setEditable(false);
                }
```

```
        }
    public static void main(String[] args){
        Consola.run(new TextFields(),375,125);
        }
    }
    class UpperCaseDocument extends PlainDocument{
        boolean uppercase=true;
        public void setUppercase(boolean flag){
            uppercase=flag;
            }
        public void insertString(int offset,String string,
        AttributeSet attributeSet)throws BadLocationException{
            if(uppercase)
                string=string.toUpperCase();
            super.insertString(offset,string,attributeSet);
            }
        }
```

Borders

JComponent contiene un método llamado **setBorder**() que permite colocar bordes en cualquier componente visible. El siguiente ejemplo muestra los distintos tipos de bordes disponibles usando el método **showBorder**().

```
//Borders.java
//<applet code=Borders.java width=500 height=300></applet>
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class Borders extends JApplet{
    static JPanel showBorder(Border b){
        JPanel jp=new JPanel();
        jp.setLayout(new BorderLayout());
        String nm=b.getClass().toString();
        nm=nm.substring(nm.lastIndexOf('.')+1);
        jp.add(
new JLabel(nm,JLabel.CENTER),BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
        }
    public void init(){
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
```

```
            cp.setLayout(new GridLayout(2,4));
            cp.add(showBorder(new TitledBorder("Titulo")));
            cp.add(showBorder(new EtchedBorder()));
            cp.add(showBorder(new LineBorder(Color.blue)));
            cp.add(showBorder(new MatteBorder(
                        5,5,30,30,Color.green)));
            cp.add(showBorder(new BevelBorder(
                                BevelBorder.RAISED)));
            cp.add(showBorder(new SoftBevelBorder(
                                BevelBorder.LOWERED)));
            cp.add(showBorder(new CompoundBorder(
                    new EtchedBorder(),new LineBorder(Color.red))));
            }
      public static void main(String[] args){
            Consola.run(new Borders(),500,300);
            }
      }
```

Los bordes pueden colocarse dentro de botones, labels, etc, todo aquello que derive de **JComponent**.

JScrollPanes.

JScrollPane funciona muy bien de manera independiente, pero también podemos controlarlo nosotros. Existen barras horizontales, verticales y combinadas.

```
//JScrollPanes.java
//<applet code=JScrollPanes width=300 height=725></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.serge.swing.*;
public class JScrollPanes extends JApplet{
      JButton
            b1=new JButton("Text Area1"),
            b2=new JButton("Text Area2"),
            b3=new JButton("Reemplazar texto"),
            b4=new JButton("Insertar texto");
      JTextArea
            t1=new JTextArea("t1",1,20),
            t2=new JTextArea("t2",4,20),
            t3=new JTextArea("t3",1,20),
            t4=new JTextArea("t4",10,10),
            t5=new JTextArea("t5",4,20),
            t6=new JTextArea("t6",10,10);
```

```
            JScrollPane
                sp3=
      new JScrollPane(t3,JScrollPane.VERTICAL_SCROLLBAR_NEVER,
                  JScrollPa ne.HORIZONTAL_SCROLLBAR_NEVER),
                sp4=
new JScrollPane(t4,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                  JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
                sp5=
      new JScrollPane(t5,JScrollPane.VERTICAL_SCROLLBAR_NEVER,
                  JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
                sp6=new JScrollPane(
                t6,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
               JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
            class B1L implements ActionListener{
                public void actionPerformed(ActionEvent e){
                    t5.append(t1.getText()+"\n");
                    }
                }
            class B2L implements ActionListener{
                public void actionPerformed(ActionEvent e){
                    t2.setText("insertado por boton 2");
                    t2.append(": "+t1.getText());
                    t5.append(t2.getText()+"\n");
                    }
                }
            class B3L implements ActionListener{
                public void actionPerformed(ActionEvent e){
                    String s="Reemplazo";
                    t2.replaceRange(s,3,3+s.length());
                    }
                }
            class B4L implements ActionListener{
                public void actionPerformed(ActionEvent e){
                    t2.insert("Insertado",10);
                    }
                }
            public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                Border brd=BorderFactory.createMatteBorder(
                    1,1,1,1,Color.black);
                t1.setBorder(brd);
                t2.setBorder(brd);
                sp3.setBorder(brd);
                sp4.setBorder(brd);
                sp5.setBorder(brd);
                sp6.setBorder(brd);
```

```
                b1.addActionListener(new B1L());
                cp.add(b1);
                cp.add(t1);
                b2.addActionListener(new B2L());
                cp.add(b2);
                cp.add(t2);
                b3.addActionListener(new B3L());
                cp.add(b3);
                b4.addActionListener(new B4L());
                cp.add(b4);
                cp.add(sp3);
                cp.add(sp4);
                cp.add(sp5);
                cp.add(sp6);
                }
        public static void main(String[] args){
                Consola.run(new JScrollPanes(),300,725);
                }
        }
```

JTextPane

Este objeto prove una gran funcionalidad como editor sin muchos problemas.

```
//TextPane.java
//<applet code=TextPane widht=475 height=425></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class TextPane extends JApplet{
        JTextPane tp=new JTextPane();
        JButton b=new JButton("Agrega texto");
        public void init(){
                b.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                for(int i=0;i<100;i++)
                                        tp.setText(tp.getText()+
                                                "blablabla\n");
                                }
                        });
                Container cp=getContentPane();
                cp.add(new JScrollPane(tp));
                cp.add(BorderLayout.SOUTH,b);
                }
        public static void main(String[] args){
```

```
                Consola.run(new TextPane(),475,425);
                }
        }
```

CheckBoxes.

Un ***checkbox*** provee una manera sencilla de seleccionar entre dos opciones. La caja usualmente tiene una "x" o simplemente está vacía. El evento del objeto **JCheckBox** puede capturarse mediante **ActionListener**.

```
//CheckBoxes.java
//<applet code=CheckBoxes width=200 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class CheckBoxes extends JApplet{
        JTextArea t=new JTextArea(6,15);
        JCheckBox
                cb1=new JCheckBox("Caja 1"),
                cb2=new JCheckBox("Caja 2"),
                cb3=new JCheckBox("Caja 3");
        public void init(){
                cb1.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                trace("2",cb2);
                                }
                        });
                cb3.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                trace("3",cb3);
                                }
                        });
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(new JScrollPane(t));
                cp.add(cb1);
                cp.add(cb2);
                cp.add(cb3);
                }
        void trace(String b,JCheckBox cb){
                if(cb.isSelected())
                        t.append("Caja"+b+"set\n");
                else
                        t.append("Caja"+b+"Limpia \n");
                }
```

```
        public static void main(String[] args){
                Consola.run(new CheckBoxes(),200,200);
                }
        }
```

Radio buttons.

Los ***radio buttons*** son asociados a un grupo de botones **ButtonGroup**. Los botones pueden estar seleccionados con el valor de **true**.

```
//RadioButtons.java
//<applet code=RadioButtons width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;

public class RadioButtons extends JApplet{
        JTextField t=new JTextField(15);
        ButtonGroup g=new ButtonGroup();
        JRadioButton
                rb1=new JRadioButton("uno",false),
                rb2=new JRadioButton("dos",false),
                rb3=new JRadioButton("tres",false);
        ActionListener a1=new ActionListener(){
                public void actionPerformed(ActionEvent e){
                        t.setText(
        "Radio Button"+ ((JRadioButton)e.getSource()).getText());
                        }
                };
        public void init(){
                rb1.addActionListener(a1);
                rb2.addActionListener(a1);
                rb3.addActionListener(a1);
                g.add(rb1);g.add(rb2);g.add(rb3);
                t.setEditable(false);
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(t);
                cp.add(rb1);
                cp.add(rb2);
                cp.add(rb3);
                }
        public static void main(String[] args){
                Consola.run(new RadioButtons(),200,100);
                }
```

```
}
```

Combo boxes.

Con JComboBox elegimos un elemento de una lista.

```
//ComboBoxes.java
//<applet code=ComboBoxes width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class ComboBoxes extends JApplet{
        String[] descripcion={ "Obtuso","Brillante",
                "Florido","Capaz","Apto","Triunfador",
                "Perdedor","Tonto"};
        JTextField t=new JTextField(15);
        JComboBox c=new JComboBox();
        JButton b=new JButton("Agregar");
        int count=0;
        public void init(){
                for(int i=0;i<4;i++)
                        c.addItem(descripcion[count++]);
                t.setEditable(false);
                b.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                if(count<descripcion.length)
                                        c.addItem(descripcion[count++]);
                                }
                        });
                c.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                t.setText("index: "+c.getSelectedIndex()
                                        +" "+((JComboBox)e.getSource())
                                                .getSelectedItem());
                                }
                        });
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(t);
                cp.add(c);
                cp.add(b);
                }
        public static void main(String[] args){
                Consola.run(new ComboBoxes(),200,100);
                }
```

```
}
```

List Boxes.

Estas listas son un poco distintas de las anteriores, no sólo en apariencia. **JList** ocupa un determinado espacio en pantalla y no cambia. Para seleccionar los elementos de la lista llamamos al método **getSelectedValues**(), ya que puede haber más de una selección.

```
//List.java
//<applet code=List width=250 height?375></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class List extends JApplet{
        String[] sabores={"Chocolate","Fresa","Vainilla",
                        "Limon","Guanabana","Melon",
                        "Frambuesa","Galleta"};
        DefaultListModel lItems=new DefaultListModel();
        JList lst=new JList(lItems);
        JTextArea t=new JTextArea(sabores.length,20);
        JButton b=new JButton("Agregar");
        ActionListener bl=new ActionListener(){
                public void actionPerformed(ActionEvent e){
                        if(count<sabores.length){
                                lItems.add(0,sabores[count++]);
                        }else {
                                b.setEnabled(false);
                                }
                        }
                };
        ListSelectionListener l1=new ListSelectionListener(){
                public void valueChanged( ListSelectionEvent e){
                        t.setText("");
                        Object[] items=lst.getSelectedValues();
                        for(int i=0;i<items.length;i++)
                                t.append(items[i]+"\n");
                        }
                };
        int count=0;
        public void init(){
                Container cp=getContentPane();
                t.setEditable(false);
                cp.setLayout(new FlowLayout());
```

```
            Border brd=BorderFactory.createMatteBorder(
                  1,1,2,2,Color.black);
            lst.setBorder(brd);
            t.setBorder(brd);
            for(int i=0;i<4;i++)
                  lItems.addElement(sabores[count++]);
            cp.add(t);
            cp.add(lst);
            cp.add(b);
            lst.addListSelectionListener(l1);
            b.addActionListener(bl);
            }
      public static void main(String[] args){
            Consola.run(new List(),250,375);
            }
      }
```

Tabbed Panes.

```
//TabbedPane1.java
//<applet code=TabbedPane1 width=350 height=200></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.* ;
import com.serge.swing.* ;
public class TabbedPane1 extends JApplet{
      String[] sabores={"Chocolate","Fresa","Vainilla",
                  "Limon","Guanabana","Melon",
                  "Frambuesa","Galleta"};
      JTabbedPane tabs=new JTabbedPane();
      JTextField txt=new JTextField(20);
      public void init(){
            for(int i=0;i<sabores.length;i++)
                  tabs.addTab(sabores[i],new JButton(""+i));
            tabs.addChangeListener(new ChangeListener(){
                  public void stateChanged(ChangeEvent e){
                        txt.setText("Tab seleccionado"
                              +tabs.getSelectedIndex());
                        }
                  });
            Container cp=getContentPane();
            cp.add(BorderLayout.SOUTH,txt);
            cp.add(tabs);
            }
      public static void main(String[] args){
```

```
            Consola.run(new TabbedPane1(),350,200);
            }
}
```

Message Box.

Las ventanas de mensajes en Java se obtienen con ayuda del objeto **JOptionPane**. Existen diversos tipos de ellas, pero probablemente las más utilizadas sean **static JOptionPane.showMessageDialog** y **JOptionPane.showConfirmDialog**. Veamos un ejemplo con varias de estas ventanas:

```
//MessageBoxes.java
//<applet code=MessagesBoxes width=200 height=150></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class MessageBoxes extends          JApplet{
      JButton[] b={ new JButton("Alerta"),
        new JButton("Si/No"),
        new JButton("Color"),
        new JButton("Entrada"),
        new JButton("3 vals")};
JTextField txt=new JTextField(15);
ActionListener al=new ActionListener(){
      public void actionPerformed(ActionEvent e){
            String id=((JButton)e.getSource()).getText();
            if(id.equals("Alerta"))
                  JOptionPane.showMessageDialog(null,
                        "Tienes un insecto","Hey!",
                        JOptionPane.ERROR_MESSAGE);
            else if(id.equals("Si/No"))
                  JOptionPane.showConfirmDialog(null,
                        "o no","escoge si",
                        JOptionPane.YES_NO_OPTION);
            else if(id.equals("Color")){
                  Object[] options={ "Rojo","Verde"};
                  int sel=JOptionPane.showOptionDialog(
                        null,"Escoge color","Advertencia",
                        JOptionPane.DEFAULT_OPTION,
                        JOptionPane.WARNING_MESSAGE,
                        null,options,options[0]);
                  if(sel!=JOptionPane.CLOSED_OPTION)
                        txt.setText("Color seleccionado: "
```

```
                        +options[sel]);
                }
        else if(id.equals("Entrada")){
                String val=JOptionPane.showInputDialog(
                        "Cuantos dedos ves?");
                txt.setText(val);
                }
        else if(id.equals("3 vals")){
                Object[] selections={ "Primero",
                        "Segundo","Tercero"};
                Object val=JOptionPane.showInputDialog(
                        null,"Escoge uno","Entrada",
                JOptionPane.INFORMATION_MESSAGE,
                null,selections,selections[0]);
                if(val!=null)
                        txt.setText(val.toString());
                        }
                }
        };
public void init(){
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i=0;i<b.length;i++){
                b[i].addActionListener(al);
                cp.add(b[i]);
                }
        cp.add(txt);
        }
public static void main(String[] args){
        Consola.run(new MessageBoxes(),200,200);
        }
}
```

Menus.

Todo componente capaz de tener un menú como **JApplet** , **JFrame**, **JDialog** y sus descendientes tienen el método **setMenuBar**() que acepta un objeto **JMenuBar**. Se agregan **JMenu** a **JMenuBar** y **JMenuItem** a **JMenu**. Cada elemento del menú es un objeto **JMenuItem** y puede tener su **ActionListener** independiente de cualquier otro elemento del menú.

```
//SimplesMenus.java
//<applet code=SimplesMenus width=200 height=75></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
```

```
import com.serge.swing.*;
public class SimplesMenus extends JApplet{
        JTextField t=new JTextField(15);
        ActionListener al=new ActionListener(){
                public void actionPerformed(ActionEvent e){
                        t.setText(((JMenuItem)e.getSource()).getText();
                        }
                };
        JMenu[] menus={new JMenu("Winken"),
                                new JMenu("Blinken"),
                                new JMenu("Nod")};
        JMenuItem[] items={
                new JMenuItem("Fee"),new JMenuItem("Fi"),
                new JMenuItem("Fo"),new JMenuItem("Zip"),
                new JMenuItem("Zap"),new JMenuItem("Zot"),
                new JMenuItem("Olly"),new JMenuItem("Oxen"),
                new JMenuItem("Free")};
        public void init(){
                for(int i=0;i<items.length;i++){
                        items[i].addActionListener(al);
                        menus[i%3].add(items[i]);
                        }
                JMenuBar mb=new JMenuBar();
                for(int i=0;i<menus.length;i++)
                        mb.add(menus[i]);
                setJMenuBar(mb);
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(t);
                }
        public static void main(String[] args){
                Consola.run(new SimplesMenus(),200,75);
                }
        }
```

Pop-up menus.

La manera más directa de implementar un **JPopupMenu** es crear una clase interna que herede a **MouseAdapter**, luego, ese agregar un objeto de esa clase interna para cada componente que produzca el ***popup***.

```
//Popup.java
//<applet code=Popup width=300 height=200></applet>
import javax.swing.*;
import java.awt.event.*;
```

```
import java.awt.*;
import com.serge.swing.*;
public class Popup extends JApplet{
        JPopupMenu popup=new JPopupMenu();
        JTextField t=new JTextField(10);
        public void init(){
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(t);
                ActionListener al=new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                         t.setText(((JMenuItem)e.getSource()).getText());
                        }
                };
                JMenuItem m=new JMenuItem("Hither");
                m.addActionListener(al);
                popup.add(m);
                m=new JMenuItem("Yon");
                m.addActionListener(al);
                popup.add(m);
                m=new JMenuItem("Afar");
                m.addActionListener(al);
                popup.add(m);
                popup.addSeparator();
                m=new JMenuItem("Quedate aqui");
                m.addActionListener(al);

                popup.add(m);
                PopupListener pl=new PopupListener();
                addMouseListener(pl);
                t.addMouseListener(pl);
                }
        class PopupListener extends MouseAdapter{
                        public void mousePressed(MouseEvent e){
                                maybeShowPopup(e);
                                }
                        public void mouseReleased(MouseEvent e){
                                maybeShowPopup(e);
                                }
                        private void maybeShowPopup(MouseEvent e){
                                if(e.isPopupTrigger()){
                                        popup.show(
                                        e.getComponent(),e.getX(),e.getY());
                                        }
                                }
                        }
```

```
        public static void main(String[] args){
                Consola.run(new Popup(),300,200);
                }
        }
```

Drawing.

En una buena **GUI** dibujar es bastante sencillo, y **Swing** tiene por supuesto esta característica. El problema con cualquier ejemplo de dibujo es que los cálculos que determinan dónde irán las cosas son típicamente más complicados que el dibujo en sí. Veamos un ejemplo con la ayuda de la función seno de la librería **Math**.

```
//SinWave.java
//<applet code=SinWave  width=700 height=400></applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import com.serge.swing.*;
class SineDraw extends JPanel{
        static final int SCALEFACTOR=200;
        int cycles;
        int points;
        double[] sines;
        int[] pts;
        SineDraw(){ setCycles(5);}
        public void setCycles(int newCycles){
                cycles=newCycles;
                points =SCALEFACTOR*cycles*2;
                sines=new double[points];
                pts=new int[points];
                for(int i=0;i<points;i++){
                        double radians=(Math.PI/SCALEFACTOR)*i;
                        sines[i]=Math.sin(radians);
                        }
                repaint();
                }
        public void paintComponent(Graphics g){
                super.paintComponent(g);
                int maxWidth=getWidth();
                double hstep=(double)maxWidth/(double)points;
                int maxHeight=getHeight();
                for(int i=0;i<points;i++)
                 pts[i]=(int)(sines[i]*maxHeight/2*.95+maxHeight/2);
                g.setColor(Color.red);
                for(int i=1;i<points;i++){
                        int x1=(int)((i-1)*hstep);
```

```
                    int x2=(int)(i*hstep);
                    int y1=pts[i-1];
                    int y2=pts[i];
                    g.drawLine(x1,y1,x2,y2);
                    }
                }
        }
public class SineWave extends JApplet{
        SineDraw sines=new SineDraw();
        JSlider cycles=new JSlider(1,30,5);
        public void init(){
                Container cp=getContentPane();
                cp.add(sines);
                cycles.addChangeListener(new ChangeListener(){
                        public void stateChanged(ChangeEvent e){
                                sines.setCycles(
                                        ((JSlider)e.getSource()).getValue());
                                }
                        });
                cp.add(BorderLayout.SOUTH,cycles);
                }
        public static void main(String[] args){
                Consola.run(new SineWave(),700,400);
                }
        }
```

Dialog Boxes.

Las ventanas de diálogo aparecen con un propósito en específico. Son muy utilizadas dentro de ambientes "***windows***". Para crear una ventana de diálogo, se hereda de la clase **JDialog**, que es precisamente un tipo de **Window**, como **JFrame**. Un **JDialog** tiene también su **layout manager** y puede manejar **listeners**. Veamos un ejemplo muy simple:

```
//Dialogs.java
//<applet code=Dialogs width=125 height=75></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
class MiDialog extends JDialog{
        public MiDialog(JFrame parent){
                super(parent,"MiDialogo",true);
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(new JLabel("Aqui esta mi dialog"));
                JButton ok=new JButton("Aceptar");
```

```
                ok.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                dispose();
                                }
                        });
                cp.add(ok);
                setSize(150,125);
                }
        }
public class Dialogs extends JApplet{
        JButton b1=new JButton("Ventana Dialog");
        MiDialog dlg=new MiDialog(null);
        public void init(){
                b1.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                dlg.show();
                                }
                        });
                getContentPane().add(b1);
                }
        public static void main(String[] args){
                Consola.run(new Dialogs(),125,75);
                }
        }
```

File Dialogs.

Todos los sistemas operatives gráficos soportan la aperture de archivos, por lo que la clase JFileChooser nos presenta una manera simple de realizar esta operación.

El siguiente ejemplo muestra dos tipos distintos de JFileChooser, una para abrir y otro para guardar documentos.

```
//FileChooserTest.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.serge.swing.*;
public class FileChooserTest extends JFrame{
        JTextField
                filename=new JTextField(),
                dir=new JTextField();
        JButton
                open=new JButton  ("Abrir"),
                save=new JButton("Guardar");
        public FileChooserTest(){
```

```
        JPanel p=new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        Container cp=getContentPane();
        cp.add(p,BorderLayout.SOUTH);
        dir.setEditable(false);
        filename.setEditable(false);
        p=new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(filename);
        p.add(dir);
        cp.add(p,BorderLayout.NORTH);
        }
class OpenL implements ActionListener{
        public void actionPerformed(ActionEvent e){
                JFileChooser c=new JFileChooser();
                int rVal=
                        c.showOpenDialog(FileChooserTest.this);
                if(rVal==JFileChooser.APPROVE_OPTION){
                        filename.setText(c.getSelectedFile().
                                getName());
                        dir.setText(c.getCurrentDirectory().
                                toString());
                        }
                if(rVal==JFileChooser.CANCEL_OPTION){
                        filename.setText("Cancelaste");
                        dir.setText("");
                        }
                }
        }
class SaveL implements ActionListener{
        public void actionPerformed(ActionEvent e){
                JFileChooser c=new JFileChooser();
                int rVal=c.showSaveDialog(FileChooserTest.this);
                if(rVal==JFileChooser.APPROVE_OPTION){
                        filename.setText(c.getSelectedFile().
                                        getName());
                        dir.setText(c.getCurrentDirectory().
                                        toStri ng());
                        }
                if(rVal==JFileChooser.CANCEL_OPTION){
                        filename.setText("Cancelaste");
                        dir.setText("");
                        }
                }
```

```
        }
    public static void main(String[] args){
        Consola.run(new FileChooserTest(),250,110);
        }
    }
```

HTML y Swing.

Cualquier componente puede tomar texto HTML, que sigue las reglas de este lenguaje. Veamos un ejemplo:

```
//HTMLButton.java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.serge.swing.*;
public class HTMLButton extends JFrame{
    JButton b=new JButton("<html><b><font size=+2>"+
            "<center>Hola<br><i>Presioname!");
    public HTMLButton(){
        b.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                getContentPane().add(new JLabel("<html"+
                "<i><font size=+4>Bam!"));
                validate();
                }
            });
        Container cp=getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b);
        }
    public static void main(String[] args){
        Consola.run(new HTMLButton(),200,500);
        }
    }
```

Sliders y Barras de progreso.

Un slider permite al usuario modificar datos a través de su movimiento. Es una barra horizontal o vertical con una pequeña "palanca". Puede ser utilizado en el control de volumen por ejemplo. Una barra de progreso muestra los datos en una situación relativa ya sea "llena a vacía" o de alguna otra manera.

```
//Progress.java
import javax.swing.*;
import java.awt.event.*;
import java.awt.* ;
import javax.swing.border.* ;
import javax.swing.event.* ;
import com.serge.swing.* ;
public class Progress extends JFrame{
        JProgressBar pb=new JProgressBar();
        JSlider sb=new JSlider(JSlider.HORIZONTAL,0,100,60);
        public Progress(){
                Container cp=getContentPane();
                cp.setLayout(new GridLayout(2,1));
                cp.add(pb);
                sb.setValue(0);
                sb.setPaintTicks(true);
                sb.setMajorTickSpacing(20);
                sb.setMinorTickSpacing(5);
                sb.setBorder(new TitledBorder("Desplazame!"));
                pb.setModel(sb.getModel());
                cp.add(sb);
                }
        public static void main(String[] args){
                Consola.run(new Progress(),300,200);
                }
        }
```

Trees.

Usar un árbol es tan sencillo como esto:

```
add(new JTree(new Object[]{"this","that","other"}));
```

Esto despliega un árbol muy simple. La API de los árboles es muy extenso, de hecho, uno de los más largos de Swing. Afortunadamente existe un punto medio, que son los predeterminados, que usualmente nos serán bastante útiles.

El siguiente ejemplo muestra árboles predeterminados.

```
//Trees.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.serge.swing.*;
class Branch{
        DefaultMutableTreeNode r;
        public Branch(String[] data){
                r=new DefaultMutableTreeNode(data[0]);
                for(int i=1;i<data.length;i++)
                        r.add(new DefaultMutableTreeNode(data[i]));
                }
        public DefaultMutableTreeNode node(){
                return r;
                }
        }
public class Trees extends JFrame{
        String[][] data={{"Colors","Red","Blue","Green"},
                         {"Flavors","Tart","Sweet","Bland"},
                         {"Length","Short","Medium","Long"},
                         {"Volume","High","Medium","Low"},
                         {"Temperature","High","Medium","Low"},
                         {"Intensity","High","Medium","Low"},};
        static int i=0;
        DefaultMutableTreeNode root,child,chosen;
        JTree tree;
        DefaultTreeModel model;
        public Trees(){
                Container cp=getContentPane();
                root=new DefaultMutableTreeNode("root");
                tree=new JTree(root);
                cp.add(new JScrollPane(tree),BorderLayout.CENTER);
                model=(DefaultTreeModel)tree.getModel();
                JButton test=new JButton("Presiona");
                test.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                if(i<data.length){
                                        child=new Branch(data[i++]).node();
                                        chosen=
(DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
                                        if(chosen==null) chosen=root;
                                        model.insertNodeInto(child,chosen,0);
                                        }
                                }
```

```
                });
            test.setBackground(Color.blue);
            test.setForeground(Color.white);
            JPanel p=new JPanel();
            p.add(test);
            cp.add(p,BorderLayout.SOUTH);
            }
    public static void main(String[] args){
            Consola.run(new Trees(),250,250);
            }
    }
```

Tables.

Como los árboles, las tables en **Swing** son vastas y poderosas. Sin embargo, es posible crear una tabla relativamente simple con **JTable**. **JTable** controla cómo será desplegada la información. Entonces, para crear un objeto **JTable** primero creamos un **TableModel** . Podemos implementar la interfaz **TableModel** pero es usualmente más sencillo heredar la clase **AbstractTableModel**.

```
//Table.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.* ;
import com.serge.swing.* ;
public class Table extends JFrame{
    JTextArea txt=new JTextArea(4,20);
    class DataModel extends AbstractTableModel{
        Object[][] data={
            {"uno","dos","tres","cuatro"},
            {"cinco","seis","siete","ocho"},
            {"nueve","diez","once","doce"},
            };
        class TML implements TableModelListener{
            public void tableChanged(TableModelEvent e){
                txt.setText("");
                for(int i=0;i<data.length;i++){
                    for(int j=0;j<data[0].length;j++)
                        txt.append(data[i][j]+" ");
                    txt.append("\n");
                    }
                }
            }
        public DataModel(){
```

```
                addTableModelListener(new TML());
                }
        public int getColumnCount(){
                return data[0].length;
                }
        public int getRowCount(){
                return data.length;
                }
        public Object getValueAt(int row,int col){
                return data[row][col];
                }
        public void setValueAt(Object val,int row,int col){
                data[row][col]=val;
                fireTableDataChanged();
                }
        public boolean isCellEditable(int ror, int col){
                return true;
                }
        }
public Table(){
        Container cp=getContentPane();
        JTable table=new JTable(new DataModel());
        cp.add(new JScrollPane(table));
        cp.add(BorderLayout.SOUTH,txt);
        }
public static void main(String[] args){
        Consola.run( new Table(),350,200);
        }
}
```

Seleccionando la apariencia.

Uno de los aspectos más interesantes de **Swing** es que podemos cambiar la apariencia. Esto permite a nuestro programa emular el exterior en varios sistemas operativos. Podemos incluso cambiar la apariencia de manera dinámica, es decir, en tiempo real. Pero generalmente se tienen dos situaciones: escogemos la apariencia del sistema operativo actual o creamos una apariencia "***multiplataforma***" que sea igual independientemente del sistema operativo.

Usualmente la apariencia ***multiplataforma*** es seleccionada sin que tengamos que modificar el código del programa. Pero si queremos que sea el sistema operativo el que determine la apariencia, se agrega típicamente al principio de **main**() las siguientes líneas de código:

```
try{
        UIManager.setLokkAndFeel(
                UIManager.getSystemLookAndFeelClassName());
```

}catch(Exception e)

El sistema se encarga de atrapar la excepción.

Veamos un ejemplo que lee desde la línea de comandos el argumento de la selección de la apariencia:

```
//LookAndFeel.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.serge.swing.*;
public class LookAndFeel extends JFrame{
        String[] choices={
                "eeny","meeny","minie","moe","toe","you"};
        Component[] samples={
                new JButton("JButton"),
                new JTextField("JtextField"),
                new JLabel("JLabel"),
                new JCheckBox("JcheckBox"),
                new JRadioButton("Radio"),
                new JComboBox(choices),
                new JList(choices),
                };
        public LookAndFeel(){
                super("Look and Feel");
                Container cp=getContentPane();
                cp.setLayout(new FlowLayout());
                for(int i=0;i<samples.length;i++)
                        cp.add(samples[i]);
                }
        private static void usageError(){
                System.out.println(
                "Uso:LookAndFeel[cross|system|motif]");
                System.exit(1);
                }
        public static void main(String[] args){
                if(args.length==0) usageError();
                if(args[0].equals("cross")){
                        try{
                                UIManager.setLookAndFeel(
UIManager.getCrossPlatformLookAndFeelClassName());
                        }catch(Exception e){
                                e.printStackTrace(System.err);
                                }
                        }
```

```
                else if(args[0].equals("system")){
                        try{
                                UIManager.setLookAndFeel(
UIManager.getSystemLookAndFeelClassName());
                        }catch(Exception e){
                                e.printStackTrace(System.err);
                                }
                        }
                else if(args[0].equals("motif")){
                        try{
                                UIManager.setLookAndFeel(
"com.sun.java."+"swing.plaf.motif.MotifLookAndFeel");
                        }catch(Exception e){
                                e.printStackTrace(System.err);
                                }
                        }
                else usageError();
                Consola.run(new LookAndFeel(),300,200);
                }
        }
```

El clipboard.

La **JFC** limita operaciones limitadas dentro del sistema de ***portapapeles***. Podemos copiar objetos **String** al ***portapapeles*** como texto, y podemos pegar texto desde el portapapeles a un objeto **String**. El ***portapapeles*** está diseñado para cualquier tipo de objetos, pero cómo lo representa en él es totalmente distinto.

El siguiente ejemplo es un simple demostración de pegar, cortar, copiar con datos **String** dentro de un **JTextArea**. Una cosa que hay que destacar es que la manera en que copiamos y pegamos normalmente también las soporta Java.

```
//CutAndPaste.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.serge.swing.*;
public class CutAndPaste extends JFrame{
        JMenuBar mb=new JMenuBar();
        JMenu edit=new JMenu("Editar");
        JMenuItem
                cut=new JMenuItem("Cortar"),
                paste=new JMenuItem("Pegar"),
                copy=new JMenuItem("Copiar");
```

```
        JTextArea txt=new JTextArea(20,20);
        Clipboard clipbd=getToolkit().getSystemClipboard();
        public CutAndPaste(){
                cut.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                String selection=txt.getSelectedText();
                                if(selection==null)
                                        return;
                                StringSelection clipString=
                                new StringSelection(selection);
                                clipbd.setContents(clipString,clipString);
                                }

                        });
                copy.addActionListener(new CopyL());
                paste.addActionListener(new PasteL());
                edit.add(cut);
                edit.add(copy);
                edit.add(paste);
                mb.add(edit);
                setJMenuBar(mb);
                getContentPane().add(txt);
                }
        class CopyL implements ActionListener{
                public void actionPerformed(ActionEvent e){
                                String selection=txt.getSelectedText();
                                if(selection==null)
                                        return;
                                StringSelection clipString=
                                new StringSelection(selection);
                                clipbd.setContents(clipString,clipString);
                                }
                }

/*      class CutL implements ActionListener{
                public void actionPerformed(ActionEvent e){
                                String selection=txt.getSelectedText();
                                if(selection==null)
                                        return;
                                StringSelection clipString=
                                new StringSelection(selection);
                                clipbd.setContents(clipString,clipString);
                                txt.replaceRange("",txt.getSelectionStart(),
                                        getSelectionEnd());
                                }
                }               */
        class PasteL implements ActionListener{
```

```
        public void actionPerformed(ActionEvent e){
                Transferable clipData=
                        clipbd.getContents(CutAndPaste.this);
                try{
                        String clipString=
                                (String)clipData.getTransferData(
                                        DataFlavor.stringFlavor);
                        txt.replaceRange(clipString,txt.getSelectionStart(),
                                txt.getSelectionEnd());
                        }catch(Exception f){
                                System.err.println("Not a String flavor");
                                }
                }
        }
public static void main(String[] args){
        Consola.run(new CutAndPaste(),300,200);
        }
}
```

Empaquetando un applet en un archivo JAR.

Una de las aplicacio nes más importantes de la herramienta JAR es optimizar la decarga de applets. Anteriormente se mencionó como se realiza esta operación.

Cambiando eventos dinámicamente.

Una de las ventajas de Swing es su modelo de eventos ya que permite gran flexibilidad. Podemos agregar y quitar eventos con llamadas a métodos.

```
//DynamicEvents.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.serge.swing.*;
public class DynamicEvents extends JFrame{
        ArrayList v=new ArrayList();
        int i=0;
        JButton b1=new JButton("Boton 1"),
                b2=new JButton("Boton 2");
        JTextArea txt=new JTextArea();
        class B implements ActionListener{
                public void actionPerformed(ActionEvent e){
                        txt.append("Un boton fue presionado\n");
```

```
                }
        }
class CountListener implements ActionListener{
        int index;
        public CountListener(int i){ index=i;}
        public void actionPerformed(ActionEvent e){
                txt.append("Contador Listener "+index+"\n");
                }
        }
class B1 implements ActionListener{
        public void actionPerformed(ActionEvent e){
                txt.append("Boton 1 presionado\n");
                ActionListener a=new CountListener(i++);
                v.add(a);
                b2.addActionListener(a);
                }
        }
class B2 implements ActionListener{
        public void actionPerformed(ActionEvent e){
                txt.append("Boton 2 presionado\n");
                int end =v.size()-1;
                if(end>=0){
                        b2.removeActionListener(
                                (ActionListener)v.get(end));
                        v.remove(end);
                        }
                }
        }
public DynamicEvents(){
        Container cp=getContentPane();
        b1.addActionListener(new B());
        b1.addActionListener(new B1());
        b2.addActionListener(new B());
        b2.addActionListener(new B2());
        JPanel p=new JPanel();
        p.add(b1);
        p.add(b2);
        cp.add(BorderLayout.NORTH,p);
        cp.add(new JScrollPane(txt));
        }
public static void main(String[] args){
        Consola.run(new DynamicEvents(),250,400);
        }
}
```

Separando la parte lógica de la parte gráfica.

En general, los diseños de clases deben ser de tal manera que cada clase cumpla con una tarea específica. Esto es particularmente importante cuando diseñamos una GUI ya que así resulta fácil diferenciar “lo que hacemos” de “cómo lo mostraremos”.

El siguiente ejemplo muestra una manera sencilla de separar el asunto lógico del código GUI.

```
//Separacion.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.applet.*;

class BusinessLogic{
        private int modifier;
        public BusinessLogic(int mod){
                modifier=mod;
                }
        public void setModifier(int mod){
                modifier=mod;
                }
        public int getModifier(){
                return modifier;
                }
        public int calculation(int arg){
                return arg*modifier;
                }
        public int calculation2(int arg){
                return arg+modifier;
                }
        }
public class Separacion extends JFrame{
        JTextField t=new JTextField(15),
                        mod=new JTextField(15);
        BusinessLogic b1=new BusinessLogic(2);
        JButton calc1=new JButton(“Calculation 1”),
                  calc2=new JButton(“Calculation 2”);
        static int getValue(JTextField tf){
                try{
                        return Integer.parseInt(tf.getText());
                        }catch(NumberFormatException e){
                                return 0;
                                }
```

```
        }

class Calc1L implements ActionListener{
        public void actionPerformed(ActionEvent e){
                t.setText(Integer.toString(
                        b1.calculation1(getValue(t))));
                }
        }
class Calc2L implements ActionListener{
        public void actionPerformed(ActionEvent e){
                t.setText(Integer.toString(
                        b1.calculation2(getValue(t))));
                }
        }

class ModL implements DocumentListener{
        public void changedUp date(DocumentEvent e){}
        public void insertUpdate(DocumentEvent e){
                b1.setModifier(getValue(mod)) ;
                }
        public void removeUpdate(DocumentEvent e){
                b1.setModifier(getValue(mod)) ;
                }
        }
public Separacion(){
        Container cp=getContentPa ne() ;
        cp.setLayout(new FlowLayout()) ;
        cp.add(t) ;
        calc1.addActionListener(new Calc1L()) ;
        calc2.addActionListener(new Calc2L() );
        JPanel p1=new JPanel() ;
        p1.add(calc1) ;
        p1.add(calc2) ;
        cp.add(p1) ;
        mod.getDocument().addDocume ntListener(new ModL()) ;
        JPanel p2=new JPanel() ;
        p2.add(new JLabel(« Modificador : »)) ;
        p2.add(mod) ;
        cp.add(p2) ;
        }
public static void main(String[] args){
        Consola.run(new Separacion(),250,100);
        }
}
```

4 Introducción a JDBC

SQL es un lenguaje usado para crear, manipular, examinar, y manejar bancos de datos relaciónales. SQL es un lenguaje de aplicación específica, por lo que un sólo "statement" o comando puede ser muy expresivo y realizar acciones de alto nivel, tales como ordenar o modificar datos.

SQL fue estandarizado en 1992, para que un programa pudiera comunicarse con la mayoría de los sistemas de bases de datos, sin tener que cambiar los comandos SQL. Desafortunadamente, debes conectarte con la base de datos, y cada base tiene una interfase diferente, así como diferentes extensiones de SQL, y estos dependen de cada proveedor. Aquí es donde entra ODBC.

ODBC es una interfase basada en C, y provee una forma consistente para comunicarse con una base de datos basada en SQL. Gracias a ODBC, es posible conectarse a una base de datos y manipularla en una forma estándar.

SQL es un lenguaje que esta bien diseñado para realizar su labor (manejar bases de datos), sin embargo, por esa misma razón, es difícil realizar con el, programas de aplicación en otras áreas, como podría ser el procesamiento de los datos para la generación de un reporte o su despliegue gráfico. Se debe recurrir a otro lenguaje de programación para realizar este tipo de aplicaciones de carácter general, típicamente C++.
Sin embargo, el lenguaje C y C++, son dependientes de la plataforma en que se compile, es decir que si un programa se compile en una PC, no correrá en una Macintosh. Y es aquí donde finalmente entra Java.

Un programa Java es totalmente independiente de la plataforma en la que se compile, por lo que puede correr en cualquier plataforma con la Java Virtual Machine (JVM) habilitada, sin tener que ser recompilado. Como sabemos, Java incluye varias librerías estándar, una de ellas es JDBC, que se puede pensar como una versión de ODBC en java. Por las ventajas de Java, JDBC esta siendo desarrollado rápidamente, y muchas compañías han creado y siguen ocupadas creando puentes entre JDBC API y sus sistemas particulares.

Por otra parte, JavaSoft, también nos ha provisto de un puente que traduce JDBC en ODBC, lo cual permite a un programa Java comunicarse a muchas bases de datos que no tienen ni la menor idea de que Java existe.

Es por eso que Java y el puente JDBC-ODBC, nos ofrece una muy buena solución al momento de escribir programas portables que requieran el uso de bases de datos, de aquí la importancia de estudiar el presente capitulo.

Empezar con JDBC

Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos.

1. Instalar Java y el JDBC en nuestra máquina.

Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el JDBC.. El código de ejemplo de demostración del API del JDBC 1.0 fue escrito para el JDK 1.1 y se ejecutará en cualquier versión de la plataforma Java compatible con el JDK 1.1, incluyendo el JDK1.2. Teniendo en cuenta que los ejemplos del API del JDBC 2.0 requieren el JDK 1.2 y no se podrán ejecutar sobre el JDK 1.1.

Podrás encontrar la última versión del JDK en la siguiente dirección: http://java.sun.com/products/JDK/CurrentRelease

2. Instalar un driver en nuestra máquina.

Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuesta máquina; no se necesita ninguna configuración especial.

El driver "puente JDBC-ODBC" no es tan sencillo de configurar. Si descargamos las versiones Solaris o Windows de JDK 1.1, automáticamente obtendremos una versión del driver Bridge JDBC-ODBC, que tampoco requiere una configuración especial. Si embargo, ODBC, si lo necesita. Si no tenemos ODBC en nuestra máquina, necesitaremos preguntarle al vendedor del driver ODBC sobre su instalación y configuración.

3. Instalar nuestro Controlador de Base de Datos si es necesario.

Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

4. Contar con una Base de Datos.

Adicionalmente, debemos contar con una base de datos compatible con ODBC que podamos usar. Se debe tener una fuente de datos ODBC “ODBC data source”. Para esto se tienen varias opciones, por ejemplo, conectarse con una base de datos Oracle o Sybase. Ponte en contacto con tu administrador si necesitas ayuda a este respecto.

Nota: Para este capitulo se requieren nociones básicas del lenguaje SQL, por lo que si tienes poca experiencia a este respecto, te recomiendo que busques un libro o un manual donde puedas consultar la notación y los comandos utilizados para crear tablas (create), insertar datos (insert) y realizar búsquedas (select). Si no lo tienes a la mano, puedes seguir con el capitulo, ya que esta explicado lo que realizan los statements de SQL utilizados, sin

embargo, para realizar tu propias aplicaciones, probablemente requerirás de más comandos y un conocimiento más profundo de las bases de datos.

Un ejemplo completo

Ver rápidamente un ejemplo, completo aunque simple, nos ayudara a captar fácilmente los conceptos de JDBC. Cuando intentamos escribir una aplicación Java, basada en una base de datos, en general, nos encontramos con los siguientes puntos:

- **Creando una base de datos**. Puedes crear la base de datos desde fuera de Java, utilizando las herramientas que incluya el vendedor de la base, o desde un programa Java que alimente la base con statements SQL.
- **Conectándose a un ODBC Data Source**. Un ODBC Data Source es una base de datos que esta registrada con el ODBC driver. En Java se puede usar tanto el puente de JDBC a ODBC, como el JDBC al puente específico del vendedor.
- **Introduciendo la información a la base de datos.** Aquí, nuevamente, se puede usar tanto las herramientas que provea el vendedor de la base, como los SQL statements que se manden desde un programa Java.
- **Obtener información de la base.** Aquí se usarán SQL statements para obtener resultados de la base de datos desde un programa Java, y luego se usará Java para desplegarlos o manipularlos.

Creando una Base de Datos

Para este ejemplo, considérese el interés por saber la cantidad de horas que asisten los programadores de la Facultad de Ingeniería al laboratorio de computación. Un reporte debe ser generado semanalmente que incluya el total de horas y la máxima cantidad de horas que pasa un programador en un día. Aquí se presentan los datos observados:

Horas de trabajo en el laboratorio de Computación, Facultad de Ingeniería

Programador	Día	# Horas
Gilbert	Lun	1
Wally	Lun	2
Edgar	Mar	8
Wally	Mar	2
Eugene	Mar	3
Josephine	Mie	2
Eugene	Jue	3
Gilbert	Jue	1

Clarence	Vie	9
Edgar	Vie	3
Josephine	Vie	4

Para este ejemplo supondremos que existe un ODBC data source con el nombre de LabComp.

Para crear esta base de datos, puedes alimentar el ODBC a través del puente JDBC-ODBC. Para ello debemos crear una aplicación Java que siga los siguientes pasos:

1.- Cargar el puente JDBC-ODBC. Debemos cargar el driver que le dirá a las clases JDBC como "hablar" con la fuente de datos. En este caso, se necesitará la clase JdbcOdbcDriver.

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

En el caso de que se trate de otro driver el que vaya a ser utilizado, la documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es **jdbc.DriverXYZ**, cargaríamos el driver con esta línea de código.

Class.forName("jdbc.DriverXYZ");

2.- Conectarse a una fuente de datos. Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos. Una URL es usada para conectarse a un JDBC data source particular.

Connection con = DriverManager.getConnection(url, "miLogin", "miPassword");

Este paso también es sencillo, lo más duro es saber qué suministrar para **url**. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con **jdbc:odbc:**. el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada "**Fred,**" por ejemplo, nuestro URL podría ser **jdbc:odbc:Fred**. En lugar de "**myLogin**" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de "**myPassword**" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre "**Fernando**" y la password of "**J8,**" estas dos líneas de código estableceran una conexión.

String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernando", "J8");

En todo caso, la documentación del driver debe de proporcionarnos las guías para el resto de la URL de la JDBC.

Si uno de los drivers que hemos cargado reconoce la URL suministada por el método **DriverManager.getConnection**, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface **Driver**, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnection**.

La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **con** es una conexión abierta, y se utilizará en los ejemplos posteriores.

3.- Mandar los statements SQL para crear la tabla. Para ello, primero hay que pedirle al objeto **conecction** (con), un objeto de tipo **statement** (stmt).
Statement stmt = con.createStatement();

Con lo que ya podemos ejecutar el siguiente statement SQL para crear una tabla de nombre LabComp:

```
create table LabComp (
  programador varchar (32),
  día char (3),
  horas integer,
  );
```

Para lo que se usa el codigo Java:

```
stmt.execute(        "create table LabComp ("+
                     "programador varchar (32),"+
                     "día char (3),"+
                     "horas integer);"
     );
```

Después de crear la tabla, podemos insertar los valores apropiados, como son:

```
insert into LabComp values ('Gilbert', 'Lun', 1);
insert into LabComp values ('Wally', 'Lun', 2);
insert into LabComp values ('Edgar', 'Mar', 8);
... etc.
```

En seguida se muestra el código completo del programa:

```
import java.sql.*;  //importa todas las clases de JDBC

public class CreateLabComp {
```

```
static String[] SQL = {
   "create table LabComp ("+
      "programador varchar (32),"+
      "dia varchar (3),"+
      "horas integer);",
   "insert into LabComp values ('Gilbert', 'Lun', 1);",
   "insert into LabComp values ('Wally', 'Lun', 2);",
   "insert into LabComp values ('Edgar', 'Mar', 8);",
   "insert into LabComp values ('Wally', 'Mar', 2);",
   "insert into LabComp values ('Eugene', 'Mar', 3);",
   "insert into LabComp values ('Josephine', 'Mie', 2);",
   "insert into LabComp values ('Eugene', 'Jue', 3);",
   "insert into LabComp values ('Gilbert', 'Jue', 1);",
   "insert into LabComp values ('Clarence', 'Vie', 9);",
   "insert into LabComp values ('Edgar', 'Vie', 3);",
   "insert into LabComp values ('Josephine', 'Vie', 4);",
};

public static void main(String[] args) {
   String URL = "jdbc:odbc:labComp";
   String username = "";
   String password = "";

   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   } catch (Exception e) {
      System.out.println("Fallo la carga del driver JDBC/ODBC.");
      return;
   }

   Statement stmt = null;
   Connection con=null;
   try {
      con = DriverManager.getConnection (
         URL,
         username,
         password);
      stmt = con.createStatement();
   } catch (Exception e) {
      System.err.println("Hubo problemas al conectarse a: "+URL);
   }

   try {
      // Ejecuta el commando SQL para crear tables e insertar datos
      for (int i=0; i<SQL.length; i++) {
         stmt.execute(SQL[i]);
```

```
        }
        con.close();
      } catch (Exception e) {
        System.err.println("Problemas con el SQL mandado a "+URL+
          ": "+e.getMessage());
      }

    }
}
```

Obteniendo información de la Base de Datos

El comando select de SQL, se utiliza para buscar información en la base de datos, mediante el metodo executeQuery de Java, aplicado este, sobre un objeto de tipo statement. Los resultados son devueltos en un objeto de tipo ResultSet, los cuales pueden ser examinados renglón por renglón usando los metodos ResultSet.next y ResultSet.getXXX.

Consideremos ahora la necesidad de obtener el número máximo de tazas de café consumidas por un programador en un día. En terminos de SQL, una forma de hacerlo es ordenar la labra por la columna horas en orden descendente. Seleccionar posteriormente el nombre del programador del primer renglón e imprimirlo.

Esto se puede realizar mediante la sentencia Java:

```
ResultSet result = stmt.executeQuery(
  "SELECT programador, horas
        FROM labComp ORDER BY horas DESC;");
```

Lo que producirá un objeto de tipo **ResultSet** con los siguiente datos:

Clarence	9
Edgar	8
Josephine	4
Eugene	3
Eugene	3
Edgar	3
Wally	2
Wally	2
Josephine	2
Gilbert	1

Gilbert	1

Podemos entonces movernos al primer renglón de la tabla mediante:

result.next();

Y extraer el nombre del programador mediante:

String name = result.getString("Programador");

Y el número de horas que estuvo:

int horas = result.getInt("horas");

Que se puede desplegar fácilmente en pantalla por medio de:

```
System.out.println("El programador "+name+
        " estuvo en el laboratorio más tiempo: "+horas+" horas.");
```

resultando en la siguiente salida:

El programador Clarence estuvo en el laboratorio más tiempo: 9 horas.

Ahora, calcular el total de horas en la semana, es sólo cuestión de sumar los datos de la columna horas

Podemos primero seleccionar la columna horas mediante:

```
result = stmt.executeQuery(
        "SELECT horas FROM LabComp;");
```

Ahora nos movemos de renglón en renglón utilizando el método next, sumando los datos, hasta que devuelva falso, indicando que ya no hay más renglones en los datos.

```
// para cada renglón con datos
horas = 0;
while(result.next()) {
        horas += result.getInt("horas");
}
```

Imprimimos el número total de horas que los programadores pasaron en el laboratorio en la semana:

System.out.println("Número total de horas: "+horas);

La salida debería de ser:

Número total de horas: 38

En seguida se muestra el programa completo:

```
import java.sql.*;
public class Reporte {
   public static void main (String args[]) {
      String URL = "jdbc:odbc:labComp";
      String username = "";
      String password = "";
       try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (Exception e) {
         System.out.println("Fallo la carga del driver JDBC/ODBC.");
         return;
      }
      Statement stmt = null;
      Connection con=null;
      try {
         con = DriverManager.getConnection (
            URL,
            username,
            password);
         stmt = con.createStatement();
      } catch (Exception e) {
         System.err.println("Problemas conectando a: "+URL);
      }
      try {
         ResultSet result = stmt.executeQuery(
             "SELECT proramador, horas FROM labComp ORDER BY horas
DESC;");
         result.next();  // se mueve a la primera fila
         String name = result.getString("programador");
         int horas = result.getInt("horas");
         System.out.println("El programador "+name+
            " estuvo más tiempo en el laboratorio: "+horas+" horas.");          result =
stmt.executeQuery(
             "SELECT horas FROM labComp;");          // para cada fila de datos
         horas = 0;
         while(result.next()) {
            horas += result.getInt("horas");
         }
         System.out.println("Total de horas: "+horas+" horas.");
        con.close();
      }
```

```
      catch (Exception e) {
        e.printStackTrace();
      }
   }
}
```

Obteniendo otro tipo de datos de un resultado (MetaData)

Ocasionalmente se necesitará de cierto tipo de información acerca de la tabla de resultados que no son los datos contenidos en ella, sino de la forma en que esta construida, es decir, el número de columnas, el tipo de dato contenido en cada columna, etc.

Por ejemplo, para la sentencia:
SELECT * from LabComp
Regresará un **ResultSet** con el mismo número de columnas y renglones que la tabla LabComp. Si no conocemos el número de columnas que contiene esta tabla de antemano, podemos utilizar metadata, para averiguarlo.

Continuando con el el ejemplo anterior, obtengamos el número y tipo de columnas que nos regresa la sentencia ya utilizada:

```
ResultSet result = stmt.executeQuery(
   "SELECT programador,
   horas FROM LabComp ORDER BY horas DESC;");
```

Los metadatos de una tabla de resultados, se pueden obtener mediante la siguiente sentencia:

```
ResultSetMetaData meta = result.getMetaData();
```

La cual regresa un objeto de tipo **ResultSetMetaData** con todos los metadatos de la tabla.

Podemos entonces determinar el número de columnas en la tabla de resultados mediante el método **getColumnCount**

```
int columns = meta.getColumnCount();
```

y después ir de columna en columna, imprimiendo su nombre y tipo de dato que contiene mediante los métodos **getColumnLabel** y **getColumnTypeName** , respectivamente.

```
int numbers = 0;
for (int i=1;i<=columns;i++) {
  System.out.println (meta.getColumnLabel(i) + "\t"
```

```
            + meta.getColumnTypeName(i));
  if (meta.isSigned(i)) { // is it a signed number?
      numbers++;
  }
}
System.out.println ("Columns: " +
  columns + " Numeric: " + numbers);
```

En seguida se muestra el código fuente del programa anterior:

```
import java.sql.*; public class JoltMetaData {
   public static void main (String args[]) {
      String URL = "jdbc:odbc:CafeJolt";
      String username = "";
      String password = "";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (Exception e) {
         System.out.println("Fallo cargar el JDBC/ODBC driver.");
         return;
      }       Statement stmt = null;
      Connection con=null;
      try {
         con = DriverManager.getConnection (
            URL,
            username,
            password);
         stmt = con.createStatement();
      } catch (Exception e) {
         System.err.println("Problemas conectando a "+URL);
      }
      try {
         ResultSet result = stmt.executeQuery(
            "SELECT programador, horas FROM labComp ORDER BY horas
DESC;");
         ResultSetMetaData meta = result.getMetaData();          int numbers = 0;
         int columns = meta.getColumnCount();
         for (int i=1;i<=columns;i++) {
            System.out.println (meta.getColumnLabel(i) + "\t"
                  + meta.getColumnTypeName(i));
            if (meta.isSigned(i)) { // is it a signed number?
               numbers++;
            }
         }
         System.out.println ("Columnas: " + columns + " Números: " + numbers);
```

```
            con.close();
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```